Programming 4

Getting to know your unit

Assessment

You will be assessed by a series of assignments set by your tutor.

Computer programs are at the very heart of modern organisations and businesses. They are vital to the delivery of products and services and they help organisations to respond to a business environment that is constantly changing. Studying this unit will transform you from a computer program user to a computer program developer who can design and program solutions to a variety of problems. You will learn to use computational thinking skills to analyse problems, identify patterns and break down complex tasks into more manageable chunks. Programming is all about problem-solving, and this unit will hone your analytical and problem-solving skills in preparation for employment or further study.

How you will be assessed

This unit will be assessed internally by a series of tasks that will be set by your tutor. Throughout this unit, you will find assessment practice activities that will help you to work towards your assessments. Completing these activities will not mean that you have achieved a particular grade, but you will have carried out useful research or preparation that will be relevant when you come to complete your final assignment.

In order for you to achieve the tasks in your assignments, it is important to check that you have met all of the Pass grading criteria. You can do this as you work your way through each assignment.

If you are hoping to gain a Merit or Distinction, you should also make sure that you present the information in your assignment in the style that is required by the relevant assessment criteria. For example, Merit criteria require you to review and justify and Distinction criteria require you to evaluate.

The assignments set by your tutor will consist of a number of tasks designed to meet the criteria in the table. The first assignment is likely to be a research-based written task that requires you to explain computational thinking skills and the principles of computer programming, while the second assignment will include practical activities such as:

- designing a software solution to meet client requirements
- developing a software solution to meet client requirements.

Assessment criteria

This table shows what you must do in order to achieve a **Pass**, **Merit** or **Distinction** grade, and where you can find activities to help you.

Pass	Merit	Distinction
Learning aim A Examine the computational thinking skills and principles of computer programming		
A.P1 Explain how computational thinking skills are applied in finding solutions that can be interpreted into software applications. **Assessment practice 4.1**	**A.M1** Analyse how computational thinking skills can impact on software design and the quality of the software applications produced. **Assessment practice 4.1**	**A.D1** Evaluate how computational thinking skills can impact on software design and the quality of the software applications produced. **Assessment practice 4.1**
A.P2 Explain how the principles of computer programming are applied in different languages to produce software applications. **Assessment practice 4.1**		
A.P3 Explain how the principles of software design are used to produce high-quality software applications that meet the needs of users. **Assessment practice 4.1**		
Learning aim B Design a software solution to meet client requirements		
B.P4 Produce a design for a computer program to meet client requirements. **Assessment practice 4.2**	**B.M2** Justify design decisions, showing how the design will result in an effective solution. **Assessment practice 4.2**	
B.P5 Review the design with others to identify and inform improvements to the proposed solution. **Assessment practice 4.2**		
Learning aim C Develop a software solution to meet client requirements		
C.P6 Produce a computer program that meets client requirements. **Assessment practice 4.2**	**C.M3** Optimise the computer program to meet client requirements. **Assessment practice 4.2**	**BC.D2** Evaluate the design and optimised computer program against client requirements. **Assessment practice 4.2**
C.P7 Review the extent to which the computer program meets client requirements. **Assessment practice 4.2**		**BC.D3** Demonstrate individual responsibility, creativity and effective self-management in the design, development and review of the computer program. **Assessment practice 4.2**

Getting Started

Many problems are solved by experienced programmers before they even touch their computer keyboard. Write down a list of the tasks and questions you think a programmer would consider when designing and building a computer program. At the end of this unit, look back at the list you have made and see whether you have missed anything, such as specific tasks or questions.

A Examine the computational thinking skills and principles of computer programming

Programming is not just a question of learning how to use the programming language that is currently fashionable or in high demand. Programming is really about learning how to solve problems by thinking in a logical fashion and understanding what a programming language is, what it can do and how it is used.

Computational thinking skills

Successful computer programming relies on you exercising your computational thinking skills. These skills will help you to investigate a problem, analyse it methodically and identify potential solutions that you can further develop into working software applications. Computational thinking skills can be understood as four separate but interlocking steps, as shown in Figure 4.1.

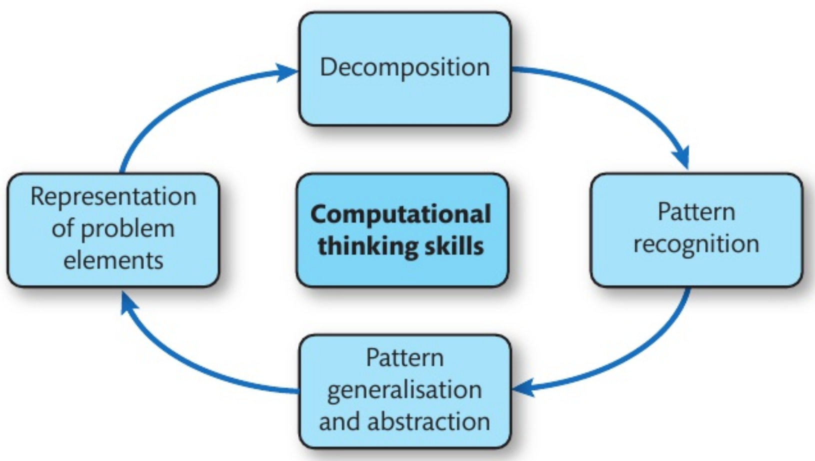

Figure 4.1: Computational thinking skills

Decomposition

Decomposition is the process of breaking complex ideas down into smaller, more manageable parts. Sometimes this process may be called factoring. Generally, problems which are not decomposed prove to be more difficult to solve. The process of breaking a larger problem down into a number of smaller problems often improves the chances of success. This is chiefly because it allows you to focus on just one thing at a time, permitting its details to be examined more closely.

Everyone uses decomposition every day, often without realising. For example, the process of making a family meal involves:

1 choosing an appropriate recipe to follow
2 calculating the correct quantity of ingredients for the recipe and the family size
3 collecting appropriate ingredients
4 preparing the ingredients
5 cooking the ingredients in the right order
6 cooking the ingredients using the correct methods
7 cooking the ingredients for the correct duration
8 assembling the meal
9 putting the meal on plates, ready to be eaten.

In this way, a single problem (making a family meal) can be decomposed into at least nine sub-tasks, each of which could be further decomposed, if necessary, until the steps required to solve each task are relatively straightforward to understand. In programming, decomposition involves the following four stages.

Identifying and describing problems and processes

At this stage, you will list problems and processes concisely, using language that matches the problem's source. For example, if you are dealing with a financial problem, you should accurately use terms from the financial sector. This means that you need to be familiar with the technical language used in the business sector relevant to the problem.

Breaking down problems and processes into distinct steps

At this stage, you will decompose complex problems and processes into separate steps which, when taken together, can be reassembled correctly. There is no specific limit to the number of steps included or of levels to which you may decompose. You will simply continue to decompose each step until you reach an acceptable level of understanding. For example, the problem of calculating someone's net pay (salary after tax is deducted) is decomposed into several steps in Figure 4.2.

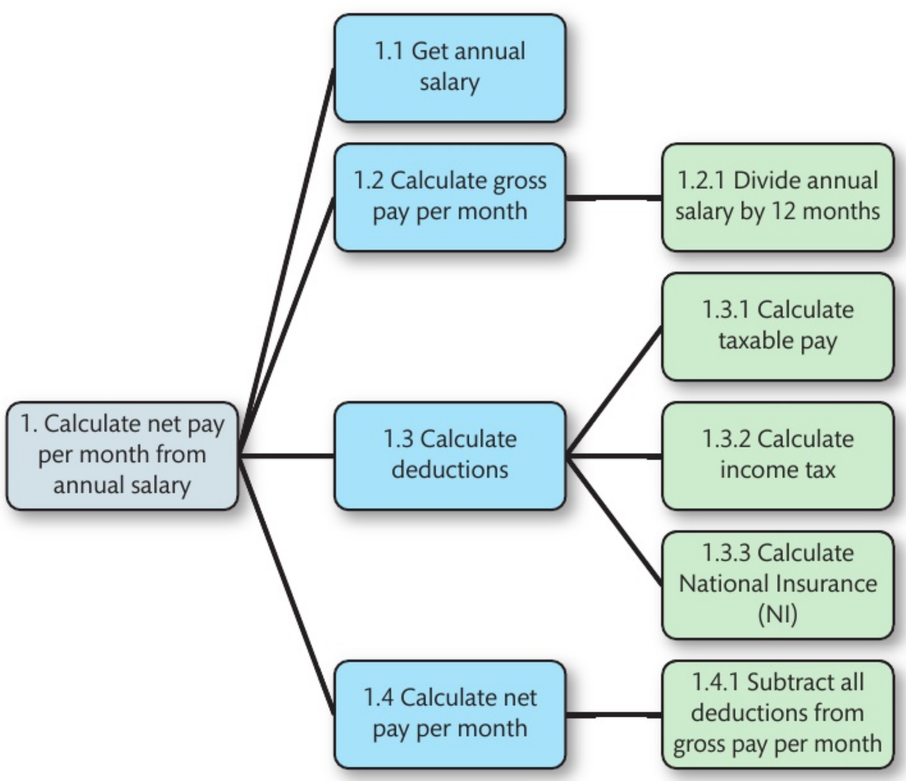

Figure 4.2: Decomposing a single process into progressively smaller steps

Describing problems and processes as a set of structured steps

At this stage, you will document the problems and processes that you have decomposed as a set of structured steps. This should be straightforward enough to be followed by you or by others.

Communicating the key features of problems and processes to others

At this stage, you will discuss the problems and processes with others. This may include other programmers or the client. Once you have decomposed a complex problem, it is possible to start looking at the steps involved to see if there are any repeating patterns.

Reflect

As a programmer, your communication skills must be flexible. You will need to adjust your delivery to meet the needs of different audiences. For example, programmers will understand technical **jargon** while a client may not. On the other hand, the client may appreciate the use of business sector-specific language, but programmers may not.

You must be able to communicate a problem to others, because having a really clear understanding of the problem is essential to your eventual success in solving it. Albert Einstein is often quoted as having said, 'If you can't explain it to a six-year-old, you don't understand it yourself.'

Key term

Jargon – words or phrases used only by a particular group, which people outside that group find difficult to understand.

Pattern recognition

Pattern recognition is the ability to see recurring attributes within the same problem and between different problems. For example, a new problem may have features which are similar to problems that have been previously encountered and solved. Recognising these repeating patterns can make problem-solving much easier, as it can provide a good place to start.

Pattern recognition is a process based on five key steps.

- Identifying common elements or features in problems or systems. This involves:
  - examining problems or systems
  - listing elements or features that exist in each
  - highlighting those which exist in multiple places
  - recognising these as patterns.
- Identifying and interpreting common differences between processes or problems. This involves:
  - examining problems and processes
  - listing elements or features that exist in each
  - highlighting those which are unique to each
  - recognising these as differences.
- Identifying individual elements within problems. This involves:
  - examining problems to identify the inputs, processes (including selections and iterations) and outputs that are present.
- Describing patterns that have been identified.
- Making predictions based on identified patterns:
  - for each identified pattern, determine how they could be used in the future or how they may appear in similar situations.

Pattern generalisation and abstraction

In computing, abstraction is a concept whereby systems are split into different layers, with each layer hiding the complexity of the layer existing beneath it. This allows a programmer to use a feature without having to know exactly how it works: the irrelevant and intricate mechanics are simply 'abstracted' away or removed.

Pattern generalisation happens when relationships between patterns can be identified and simple conclusions can be drawn. For example, patterns can be identified even when, at first, it does not look like there are many similarities.

In forms of transport, for example, cars, buses and lorries, we can see similar elements repeating in the design patterns. Each vehicle includes four wheels, two axles, a chassis, a steering mechanism and so on. In order to problem-solve effectively, you must look beyond the obvious (in this case, the physical differences between the three forms of transport). Instead, try to identify the elements that are common and the relationships between these elements.

There are two parts to this phase of computational thinking.

First, identify the information required to solve an identified problem. You can achieve this by:

- knowing what information you need
- knowing your reasons for needing this information (the 'rationale')
- knowing the format in which this information needs to be provided
- knowing how soon this information is required to prevent the solution from being delayed.

Second, filter out the information not required to solve an identified problem. You can achieve this by:

- knowing what information is not needed, as this will be a distraction
- knowing (and justifying) why you have excluded this information.

Representing parts of a problem or system in general terms

To do this, you need to identify the:

- variables – these are the values in a problem or system that may change, typically input by the user or as the result of a required calculation
- constants – these are the values in a problem or system that do not change often or that remain fixed for a reasonable period of time (e.g. the base rate of income tax being 20%)
- key processes – these are the processes that are absolutely critical to understanding a problem or how a system works
- repeated processes – these are processes that occur multiple times within a problem
- inputs – these are the values entered into the system, including the units used and, potentially, any valid values or ranges (e.g. where gender is 'M' for male or 'F' for female, or where a house price has to be between £20,000 and £2,000,000)
- outputs – this is information presented to the user in a required format, generally specified by the client as part of their requirements.

Discussion

In small groups or pairs, think about an everyday process such as calculating the cost of decorating a room with fresh paint. Try to determine the variables, constants, key processes, inputs and outputs that are involved.

Uses of software applications

Software applications (often called 'applications' or 'apps') are programs that have been developed to carry out specific tasks, solve particular problems and fulfil identified user needs. There are many different ways to categorise applications: by type of software licence (free, commercial etc.), by computer platform (desktop, mobile etc.) or by use. Table 4.1 shows some software applications categorised by popular use, including the **implications** of their use.

Key term

Implications – the likely effects of something.

Table 4.1: Uses and implications of software applications

Usage category	Needs which are fulfilled	Examples	Implications
Gaming	Videogames which can be used to entertain, educate or help recuperate after trauma.	Activision Call of Duty Bioware Mass Effect Ubisoft Assassin's Creed PopCap Peggle Microsoft Minecraft	Potential social isolation Health issues (has an impact on physical exercise) Mental wellbeing (as relaxation tool) Potential addiction
Entertainment	Applications that help users relax and enjoy various forms of media, e.g. music, video, books through downloading, streaming or both.	Apple iTunes Microsoft Media Player BBC iPlayer Amazon Kindle App	Potential social isolation *Ad hoc* viewing/listening (more flexible consumption)
Productivity	Applications such as spreadsheets, databases, word processors and presentation software which help workers complete tasks more efficiently, typically when working in administrative roles.	Microsoft Office Apache OpenOffice Adobe Creative Suite Google Drive/Apps for Work	Improved productivity New work skills New ideas and problem-solving techniques Greater efficiency Reduced costs
Information storage and management	Applications used to store and manage information safely (preventing loss) and enable rapid retrieval, typically via the internet.	Dropbox Google Drive Apple iCloud	Less risk of data loss Redundancy of data Flexible access to data
Repetitive or dangerous tasks	Applications used to automate equipment in environments which are hazardous to people or which replace monotonous manual tasks.	Energy industry Self-driving trucks (mining) Car manufacture Chemical processing	Reduced physical risk to individual More job satisfaction (fewer menial, repetitive jobs) Redundancy New work skills Improved productivity Reduced costs
Social media	Applications designed to connect you to other people, to aid communication and share ideas, events, pictures and videos.	Twitter™ Facebook™ Snapchat™ WhatsApp™ Pinterest™ Instagram™ WordPress™ Blogger™	Changes to communication skills Ability to discuss and share problems with others Ability to exchange ideas with others Security risks Risks to minors and vulnerable adults Potential addiction Improved contribution to important issues
Search engines	Applications that provide a facility for users to quickly and accurately search the world wide web for specific terms, returning meaningful results.	Google Microsoft Bing Yahoo! Search	Improved access to information Products and services found faster Less time wasting Risk of tracking and/or targeted advertising

Theory into practice

Identify 10 software applications that you may have installed on your home computer, tablet or smartphone.

- Which usage category do these applications belong to?
- What problems do they solve?
- What are the implications of using this type of software?

Features and characteristics of programming languages

Hundreds of different programming languages have been developed since the mid-20th century. Original programs were written in machine code (**binary**) instructions that told the computer's processor exactly what to do, but this proved to be time consuming and very demanding on the programmers' skills. Although fast and efficient in execution, such programs took a relatively long time to create and complicated applications were considered a major undertaking.

Over time, newer **low-level** and **high-level languages** made programming a more understandable process. As a consequence, they reduced the production time of more intricate applications and systems software. Table 4.2 contrasts two different types of program code being used to output the same message screen, initially using low-level machine code and then using the high-level language, C++.

Table 4.2: Outputting 'BTEC student' using low- and high-level program code

Low-level language	High-level language

'BTEC student' in Intel x86 machine code (shown in hexadecimal):
`B4 09`
`BA 09 01`
`CD 21`
`CD 20`
`42 54 45 43 20 73 74 75 64 65`
`6E 74 24`
	'BTEC student' in C++ source code:
	`cout << "BTEC student";`

While we can probably read the C++ code more comfortably, a computer can process the low-level equivalent much more easily as its machine code instructions talk directly to its **central processing unit (CPU)**. In comparison, the C++ code has to be successfully translated into machine code using a **compiler** before it can be executed by the CPU.

Uses and applications of high- and low-level languages

A computer's architecture is complex in construction. Low-level programming languages such as machine code (often written in binary, which is a base-2 system, or hexadecimal, which is a base-16 system) or assembly language hide little of this complexity from the programmer. In order to program the computer at this level, the developer must know the processor's architecture very well.

In comparison, high-level languages such as Microsoft Visual Basic .NET and C++ use abstraction to hide the complexities of the architecture from the programmer. In these languages, a single command may translate to hundreds of complex low-level instructions which the processor can understand.

Key terms

Binary – a number system that only uses the digits 0 and 1 to form numbers (also known as 'base 2'). For example, '5' in binary is 101 ($1 \times 4 + 0 \times 2 + 1 \times 1$). Computer circuits have 'on' and 'off' states which can be used to represent binary 0s and 1s.

Low-level and high-level languages – in programming, the terms 'low' and 'high' refer to a language's position between being understood by a computer (e.g. binary is low-level) and understood by a person (e.g. natural languages such as English are high-level).

Central processing unit (CPU) – a computer's central 'brain'. Typically it controls the computer's resources, inputs and outputs and, most importantly, the processing instructions and data fetched from its random access memory (RAM).

Compiler – a special program which translates program code written in a high-level language into binary instructions which the CPU can process.

The majority of commercial programming languages used in the world today are high-level. This is because high-level languages:

1 improve programmers' productivity when writing program code
2 improve the readability of code
3 produce code that is easier to **debug**
4 allow for the use of more flexible program development tools
5 produce code that can be **ported**.

Key terms

Debug – the process of identifying an error (or bug) in a program code and removing it.

Ported – written using one computer architecture, but compiled for use on another. This is also commonly known as cross-compiling.

Research

The term 'bug' is popularly attributed to computer pioneer Grace Hopper who, in 1945, located a moth which had become trapped between the contact points in a computer's relay switch. Once she had removed the 'bug', the computer worked again. In reality, though, the term is even older than this. Do some research to see if you can identify its origin.

Low-level languages provide the ultimate control over computer hardware. However, they are time consuming and complex to use. In contrast, high-level languages abstract the difficulties of talking to the computer hardware directly and offer more rapid software development opportunities. However, the finished program is often less efficient, slower and bigger than it might be if produced in a low-level language.

Programming paradigms

Different types of problem have spawned different programming styles. Each style, known as a paradigm, aims to solve a problem in a different way, often to fulfil different user needs. Table 4.3 shows the most common types of programming paradigm.

Some programming languages can belong to multiple paradigms. For example, Ruby is commonly used as a scripting language, but it also has many features which are object-oriented. Microsoft's Visual Basic .NET, which many classify as an event-driven language, is also heavily reliant on the classes (e.g. buttons, forms and dialogs) that represent the core of the Microsoft Windows operating system. Consequently, it can also be considered OO.

Table 4.3: Different programming language paradigms

Programming paradigm	Programming languages
Procedural	All programming languages are procedural

Features and characteristics

Procedural languages are often the first one learnt by a programmer. They are often considered to be a general-purpose tool and are used to create many different types of application. They are typically written as a set of well-defined steps which solve a set problem, e.g. performing a simple calculation in C, as shown in Figure 4.3.

If the steps become over-complicated, lengthy or repetitive, the programmer may choose to divide the steps into separate procedures, each with a single purpose that can be used multiple times.

The term 'imperative' can also be used to describe this type of language, but imperative languages tend to rely less on procedures.

```
#include <stdio.h>
#include <string.h>

int main()
{
 int a;
 int b;
 int c;

 puts("Enter first number");
 scanf("%d", &a);
 puts("Enter second number");
 scanf("%d", &b);

 c = a + b;
 printf("%d + %d = %d",a,b,c);

 return 0;
}
```

Figure 4.3: C program code displaying the sum of two inputted numbers

Programming paradigm	Programming languages
Object-oriented (OO)	• C++ • Microsoft C# • Oracle Java • PHP • Ada

Features and characteristics

Object-oriented programming is a popular modern approach to programming. OO languages rely on the concepts of 'classes' and 'objects' to solve real-world problems. Due to this approach, they are popularly used to design videogames, e-commerce websites, database systems and user interfaces.

In OO, objects are created from classes which are usually modelled on real-world 'things' such as customers, bank accounts, products, orders etc. Each class acts as a software blueprint, encapsulating (or containing) the thing's state (its data or properties) and behaviour (its functions or methods) in program code (see Figure 4.4). The programmer creates specific interactions between different objects in order to solve the problem. Because each class exists separately, they can easily be modified or adapted to reflect changes happening in the real world without having a negative effect on the whole solution. This makes OO languages very attractive to developers when they consider the demands of tackling ongoing maintenance.

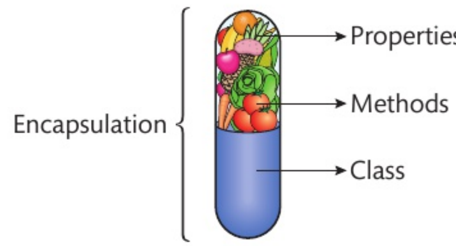

Figure 4.4: Encapsulation

Programming paradigm	Programming languages
Event-driven (ED)	• Microsoft Visual Basic .NET, Pythion, Ruby, Java

Features and characteristics

This is a popular paradigm for the development of graphical applications which respond to events generated either by the system (e.g. system clock) or the user (e.g. a mouse click).

Event-driven programs typically work non-sequentially with users able to select the operations they want to perform rather than follow the preset inputs of a more rigid program structure.

Developers typically focus on programming event handlers, which is the code that specifies the actions to perform when a particular event is triggered via a listener. A listener is a process that waits for a certain event to happen. For example, if a File->Open menu option is clicked, a file open dialog will be displayed.

Programming paradigm	Programming languages
Machine	• Machine code • Assembly language (sometimes informally referred to as 'assembler')

Features and characteristics

These are the lowest level programming languages which offer control over the basic hardware of the machine. Machine code must be written for a specific CPU family, such as Intel x86 (32-bit) or x64 (64-bit). It cannot be run easily on different platforms without conversion to the new platform's CPU instructions or emulation of the original CPU's instructions.

Assembly language uses a series of people-friendly mnemonics to represent basic CPU instructions. Mnemonics are memory aids that help people to remember complex concepts, typically through visual representation or easily-remembered sayings or rhymes. In assembly language, mnemonics improve readability and increase productivity.

For example, when adding 4 + 2 in assembly language:

- mov means move, add means add and int means interrupt (int 20 terminates the program)
- ax and bx are registers (high speed memory areas) inside the CPU.

```
mov ax, 0004
mov bx, 0002
add ax, bx
int 20
```

These mnemonics come in the form of opcodes and operands.
Opcodes describe the operation being performed and operands are the values being processed by the operation.

In machine code, this same code expressed as hexadecimal looks like this:

```
B8 04 00
BB 02 00
01 D8
CD 20
```

In order to execute, assembly language needs to be translated to machine code using a special program called an assembler.

Machine languages are often used when speed is vital or when low-level access to computer hardware or communication with connected electronics is important. For example, it may be used in vending machines, head-up displays and videogames.

Programming paradigm	Programming languages
Markup	• HTML (Hypertext Markup Language) • XML (Extensible Markup Language)

Features and characteristics

Markup is a form of language used to specify the content formatting of a document in a structured manner using special tags. For example, in HTML <p> is used to denote the start and end of a paragraph, as shown in Figure 4.5.

```
<p>A new paragraph</p>
```

Figure 4.5

- Other markup languages can be used to represent complex data structures in a platform-neutral manner, such as entries from a CD library stored in XML, as shown in Figure 4.6.
- XML is often used as the preferred format to transfer data between different computer systems and applications, such as when exporting and importing data between relational database systems that are normally incompatible.

```
<CD>
 <TITLE>Let's Dance</TITLE>
 <ARTIST>Bowie, David</ARTIST>
 <YEAR>1983</YEAR>
</CD>
<CD>
 <TITLE>Rubber Soul</TITLE>
 <ARTIST>The Beatles</ARTIST>
 <YEAR>1965</YEAR>
</CD>
```

Figure 4.6

Programming paradigm	Programming languages
Scripting	• Perl • JavaScript • Ruby • PHP

Features and characteristics

Different types of scripting languages are used for different purposes.

JavaScript is used in web design to automate processes and add interactive features to web pages.

PHP is a popular server-side scripting language used to create complex online applications used in e-commerce.

Perl and Ruby are often used to automate system processes on a computer by linking and executing tasks that may have originally been run separately by a user. For example, Figure 4.7 shows a Ruby program to display the first 255 bytes of a specified file on screen.

```
#!/usr/bin/ruby

filename = ARGV[0]

aFile = File.new(filename,"r")

if aFile
  content = aFile.sysread(255)
  puts content
else
  puts "Unable to open file!"
end
```

Figure 4.7

Comparing and contrasting programming languages

There are many factors to compare and contrast when considering different programming languages. These include the requirements (e.g. hardware, software and special devices), performance and ease of development.

Discussion

In small groups or pairs, discuss why you think you might need to compare programming languages and why it is important to have a variety of different languages. Write down the reasons you can think of. Now read on and see how many reasons you correctly identified.

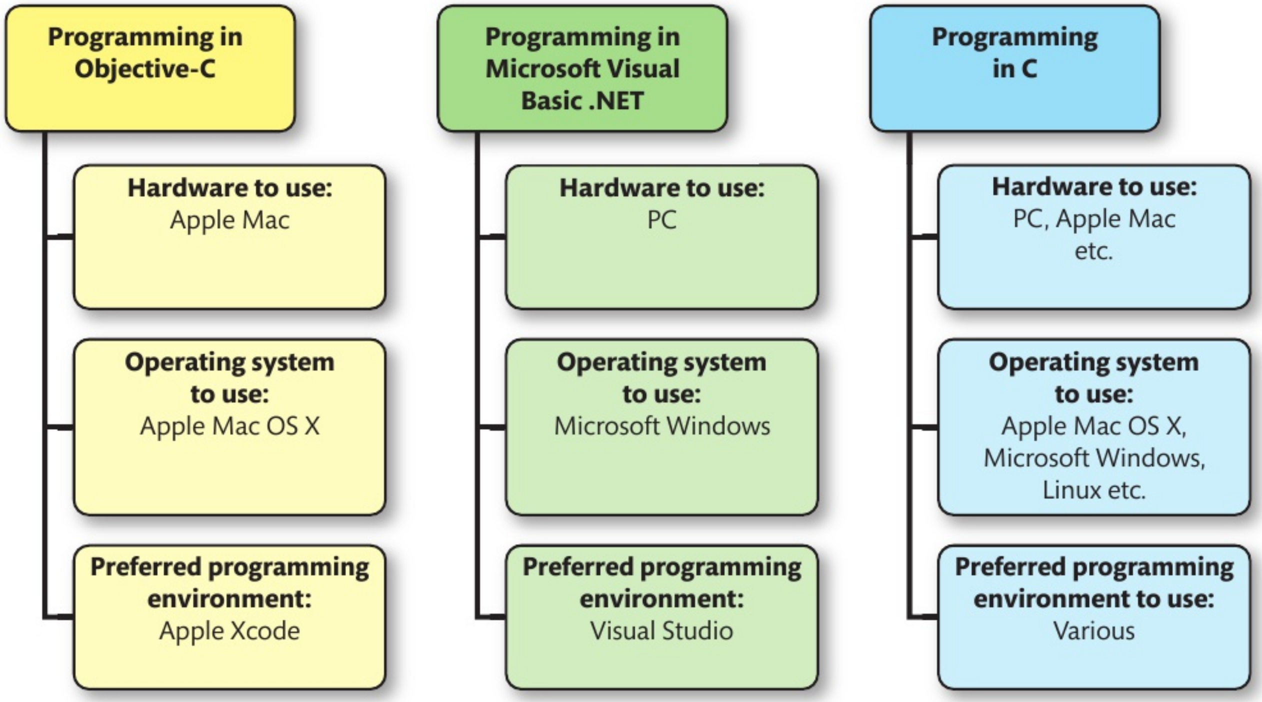

Figure 4.8: Contrasting hardware and software requirements for selected programming languages

Hardware and software needed for running and developing a program

Some programming languages require specific hardware and software for running and developing a program. Preferred requirements for popular programming languages are shown in Figure 4.8.

Some programming languages, such as C, are considered to be 'cross-platform'. This means that it is supported on many different combinations of hardware and operating system. This is likely to explain why it is still so popular with commercial programmers, despite the fact that it was initially released back in 1972.

Special devices required

When programming solutions for external hardware, it is quite common to connect a desktop PC to a special device, typically a Serial Programming Interface (SPI) or Joint Test Action Group (JTAG). This special device is usually connected to the desktop PC via a universal serial bus (USB), serial or parallel cable, and it allows the developer to perform a process called in-system programming (ISP). ISP allows devices to be reprogrammed without being removed from their original circuit. For example, a common use of ISP is to re-program a mobile telephone in order to 'unbrick' (repair) it or unlock its features, as shown in Figure 4.9.

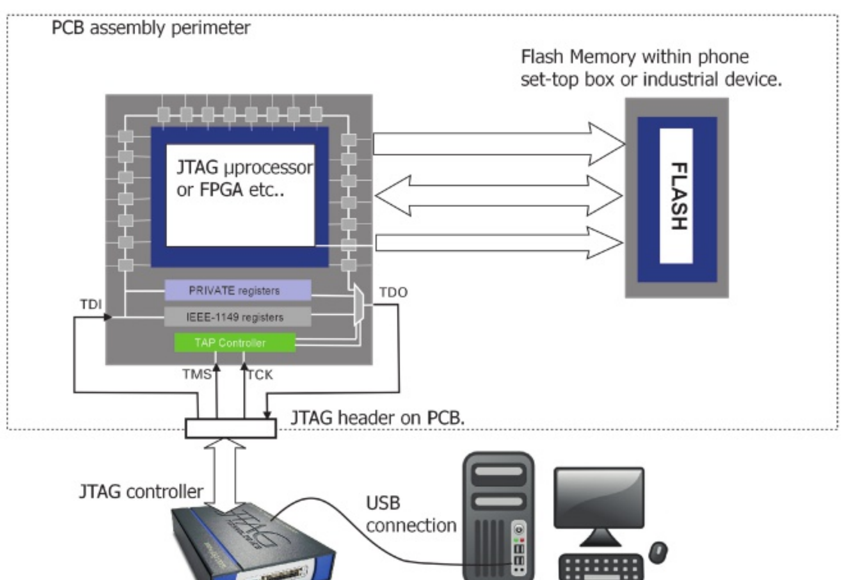

Figure 4.9: JTAG interfacing a mobile telephone and a PC

Performance

Programming languages can perform more or less efficiently than one another. Typically, performance is measured by a language's ability to execute complex algorithms against a clock. For example, some programming languages make better use of the CPU by having more efficient translators that generate tightly optimised machine code.

Some programming languages manage RAM more efficiently than others by having aggressive **garbage collection** measures. Garbage collection is not a mandatory process and some languages, such as C, require the programmer to remember to manually deallocate memory which is no longer in use. Aggressive garbage collection may make the language faster but runs the risk of memory 'leaks' occurring.

Key term

Garbage collection – an automated process which attempts to reclaim RAM reserved by a program to store data, e.g. an identifier (variable or object) that is no longer needed.

Preferred application areas

Some programming languages are better suited for some areas of application.

- C is considered to be a general-purpose language, but it is particularly effective for controlling external hardware and electronics because of its low-level control of the computer system.
- Java is used to create applets for webpages and Android mobile apps. Its 'write once, run anywhere' (WORA) approach means it is incorporated into many different forms of home entertainment device, e.g. TV set-top boxes and Blu-ray players.
- PHP is used to create server-side applications for businesses in e-tail, e-commerce etc.
- C# is used to create videogames for Microsoft Windows and popular games consoles, particularly because of its ease of use with Microsoft's XNA, which is a popular set of tools used in videogame development and management.
- Objective-C is used to create mobile apps for Apple's iOS devices, i.e. iPad, iPhone etc.

Development time

Development time is an important consideration when choosing a programming paradigm. For example, as we have seen, programs written using machine code and assembly are efficient and execute very quickly, but development time when using these languages can be much longer than when working with high-level languages in modern integrated development environments.

In commercial programming, time is equated directly to money, so there is a strong emphasis on writing reliable code with a minimum of bugs as quickly as possible. As a consequence, programming languages supported by feature-rich development tools which claim to reduce development time are popular.

However, it can be difficult to measure development time. This is because acceptable **metrics** are hard to determine and agree upon within the industry. For example, you could consider the number of lines of code (LOC) written per hour to be an acceptable metric, but this may not consider the number of related bugs that are generated. In addition, if a programmer knows that their productivity is measured by the number of LOC that they create in an hour, this may encourage them to write overly **verbose** code.

Key terms

Metric – an agreed form of measurement that enables comparison and evaluation.

Verbose – using more words or code than necessary.

Ease of development

As noted earlier, some programming languages are easier to use than others. This is because some languages present the software developer with advanced tools that can offer hints or auto-generated program code. For example, Microsoft Visual Basic .NET converts input forms created using drag-and-drop functionality into program code automatically.

The quality of other tools available within a programming language editor can also have an impact on the ease of development and a programmer's productivity. These tools include help systems, syntax highlighting and debugging tools.

- Syntax highlighting is a feature in many programming language editors which displays different programming constructs in specific colours. For example, a comment may appear in green, a string of text in red and so on. Most developers find the resulting code easier to read, understand and debug.
- Debugging tools offer the programmer a range of features that help them to identify bugs in their code and remove them. Popular debugging tools include traces, watches and breakpoints.

Research

You can research and explore different programming languages online through the use of virtual development environments. These environments let you select a particular language, key some basic program code and execute it. You will be able to find examples through an internet search.

Reflect

No matter which programming language, platform or development tools you use, you will need to develop and demonstrate professional behaviour that is appropriate for the programming industry. This includes:

- being disciplined – you should set relevant development targets and work to realistic timescales
- respecting others and their opinions, especially when they are giving you essential feedback on your program and its suitability when assessed against the original user needs
- evaluating outcomes to help you justify your recommendations and decisions, particularly in terms of the design or implementation of your solution
- comparing and contrasting set targets with their final outcomes, as this will help you to understand and improve your performance as a developer and problem solver
- working with others as part of a team – this is especially important when contributing to a larger problem-solving task which involves others. You should provide your colleagues with support, help them to manage your joint workload, respond positively and constructively to objections and conflict, and help to manage the expectations of your client.

Programming constructs and techniques and their implementation in different languages

Programming language constructs are the building blocks used to create a program. You will use them in many different ways and apply various techniques to solve the problems you are given. One thing you will discover very early on is that many constructs and techniques are common across a number of languages, which means that your knowledge can be easily transferred between languages.

Code samples from a number of different popular programming languages have been used to help you compare and contrast their constructs and techniques. However, the larger code samples in this chapter have been written in Microsoft C#, which is a very popular language within the software development industry. As your skills progress, you should find it relatively simply to convert these code samples to another target language.

Command words

The concept of command (or reserved) words is at the core of most programming languages. These reserved words cannot be used by a programmer to name things (see Identifiers). Instead, they are used to command a specific action, such as 'open a file', 'clear the screen' or 'put text on the screen'. The quantity and purpose of reserved words can vary greatly between different programming languages but, as you become more experienced as a developer, you will start to identify recurring concepts.

Link

A list of reserved words for Java is shown in *Unit 7: Mobile Apps Development*.

Identifiers

An identifier is a programmer-friendly name which represents a quantity stored in the computer's RAM. Many types of identifier are used in programs, but some of the most common types are:

- a constant – this is an identifier representing a value that will not change while the program is running
- a variable – this is an identifier representing a value that may change while the program is running.

For example, when a person logs into a program, their name and password would typically be stored as variables, as these details would be different for each different user. However, a program calculating the price of a new television in a shop's electronic till would need to ensure that the current rate of value added tax (VAT) is added to the price. This value would not change for every sale, so it could be set as a constant.

In order to create (or declare) a constant or a variable, you must provide the programming language with two things:

- a name
- a data type.

You should always choose a sensible and meaningful name. For example, a good variable name for storing the user's name would be 'username'. Some names cannot be used because they are reserved by the language for a particular use, typically because they are a command word. These reserved words vary between different programming languages, so be careful.

There are many different naming conventions used in programming. CamelCase or snake_case are perhaps the two most popular in modern programming, for example if we want to store a user's age in years in C#:

camelCase	int userAge;	The first word is in lower case. The initial letter of each successive word is capitalised.
snake_case	int user_age;	All words are in lower case. Words are linked with an underscore symbol.

Most programming languages also contain the concept of local and global variables. The easiest way to think about this is to remember that global variables can be used *anywhere* in your program code. Local variables are *limited* to being used in the block of code (e.g. a function) in which they are declared.

Link

This also applies to mobile apps development, as shown in *Unit 7: Mobile Apps Development*.

Data types

Almost all programming languages support the concept of data types. A data type is used to define what kind of value a variable or constant can store, what operations can be performed upon it and its behaviour within the program. Most programming languages offer many different data types for the programmer to use.

The most common data types used are:

- character – this stores a single character (a character is a letter of the alphabet, a digit, a blank space or a punctuation mark)
- string – this stores a collection of characters
- integer – this stores a whole number (i.e. a number without decimal places)
- real (sometimes called floating point) – this stores a number with decimal places
- **Boolean** – this is a logical value which stores either true (1) or false (0).

Key term

Boolean – a form of logical data type named after the 19th century mathematician, George Boole.

Because these data types are quite common, it is possible to compare their implementation in different programming languages, as shown in Table 4.4.

Some languages are said to be strongly typed while others are weakly (or loosely) typed. This makes a fundamental difference to the formality of the programming language. Strongly typed programming languages require data and data types to be consistent when used.

For example, in Java, if a variable has been declared an integer, it should only be used to store integer variables, otherwise errors may occur. This is because Java is a strongly typed language. However, C is loosely typed: it will implicitly convert the data to the correct data type that the variable expects without the programmer needing to add any specific code to do so. This is a convenient feature, but it can lead to unexpected results.

Tip

Be careful when dealing with data types. Always think about the data you want to store and which data type is the most appropriate to use. Getting the data type wrong can lead to problems such as truncation of decimal places and incorrect rounding, both of which could be disastrous in an application dealing with money.

Statements

Statements are the core aspect of many programming languages. They define the basic actions that can be performed and often combine a number of language constructs.

Common examples of basic statement types with direct language equivalents are shown in Table 4.5. There are many similarities between them. For example, they are all are very **case-sensitive**, but only some of them require semi-colons to mark the end of a statement.

Key term

Case-sensitive – when a programming language recognises the difference between upper-case and lower-case characters, such as 'a' and 'A'. For example, if a command word is expected in lower case, an error will occur if it is unexpectedly written in upper case. Most modern programming languages prefer lower case.

Table 4.4: Variable declarations using different data types in different programming languages

	Programming language implementation in...		
What to declare...	**C**	**C++**	**Microsoft Visual Basic .NET**
Initial (**character**)	`char initial;`	`char initial;`	`Dim initial As Char`
Username (**string**)	`char username[20];`	`string username;`	`Dim username As String`
Age (**integer**)	`int age;`	`int age;`	`Dim age as Integer`
Price (**real**)	`float price;`	`float price;`	`Dim price as Double`
Valid (**Boolean**)	Not available	`bool valid;`	`Dim valid as Boolean`

Table 4.5: Statement types, their purposes and expressions in different languages

Statement type	**Purpose**		
Assignment	Store a value in an identifier (variable or constant).		
	Microsoft C#	**C++**	**Ruby**
	`userAge = 17;`	`userAge = 17;`	`userAge = 17`
Input	Allow input from the user, typically from the keyboard or a file.		
	Microsoft C#	**C++**	**Ruby**
	`userName = Console.ReadLine();`	`cin >> userName;`	`userName = gets`
Output	Generate output for the user, typically to the screen, a printer, a speaker, a file and so on.		
	Microsoft C#	**C++**	**Ruby**
	`Console.WriteLine("Hello");`	`cout << "Hello";`	`puts "Hello"`

Control structures

The algorithms that control programs are typically built using a combination of three basic programming building blocks known as control structures. These control structures are **sequence**, **selection** and **iteration**.

Key terms

Sequence – one action after another, none missed, none repeated.
Selection – actions chosen based on a supplied condition which is evaluated to true or false.
Iteration – actions repeated a number of times, typically until a condition is no longer true.

Link

The C# code for selections and iterations is very similar to the Java examples found in *Unit 7: Mobile Apps Development*.

Logical operations

Operators are special symbols used to perform special tasks in a program. Logical operations such as 'And', 'Or', 'Not' etc. operate on Boolean principles. They are used to combine conditions in 'if' statements and various loops. For example, a logical 'And' operator (&&) is shown in the C# 'if' statement condition in Figure 4.10. In this example, a customer will get free shipping if their order is worth more than £50 and if their address is less than 25 miles. Both parts of the condition must be true to qualify for free shipping.

Care should be taken when moving between languages. For example, in some languages (e.g. Visual Basic .NET) the single '=' sign is used to test for equality, while in other languages (Java, C, C++, C#, PHP) the '==' is used instead. Take the time to investigate the different types of operators available in your target programming language.

```
if (customerOrder > 50 && deliveryDistance < 25)
{
    Console.WriteLine("You have qualified for free shipping.");
}
```

Figure 4.10: C# if statement using a logical And (&&) operator

Link

A detailed list of common Java operators (including logical types) is provided in *Unit 7: Mobile Apps Development.*

Subroutines, functions and procedures

Subroutines, functions and procedures are terms used in procedural programming, where code is split into a number of different modules. Each module may be called a subroutine, function or procedure depending on the programming language used. Each module is responsible for performing a single task and is typically somewhere between 5 and 50 lines of code in length.

The C# code shown in Figure 4.11 demonstrates the use of a programmer-defined function to calculate the area of a circle when given a specific radius.

```
class Program
{
    const float PI = 3.14f;

    static void Main(string[] args)
    {
        float radius;
        float area;

        radius = 10.0f;
        area = calcAreaCircle(radius);  // function call

        Console.WriteLine("Area of circle is {0}", area);
        Console.ReadKey();
    }

    //function declaration
    public static float calcAreaCircle(float radius)
    {
        return PI * radius * radius;
    }
}
```

Figure 4.11: Code to calculate the area of a circle given the radius

The use of these modules means that code tends to be easier to write, read and debug. Code written in this way can usually be reused through multiple solutions and allows a single application to be divided and worked on by multiple programmers simultaneously, which means that development time can be reduced.

Tip

Almost all programming languages have library functions which programmers can use when solving complex problems. These allow the programmer to perform common but crucial tasks on data, such as formatting its appearance, finding the length of a string or the square of a number. You can also download and install third-party functions from reputable websites to expand programming languages.

Data structures

A data structure is a programming technique used to collect and organise data items in a formal structure, which helps the data to be processed efficiently. Although the availability of certain data structures vary between different programming languages, many are very common. These common data structures include string, array (one- and two-dimensional), stack, queue and record.

Sometimes, it is possible to program more efficiently through the selection and use of specific data structures, especially when this is combined with iteration control structures (loops). Software developers become familiar with different data structures as they learn about different programming languages. Here are some of the most common data structures.

String (or text)

This is a data structure which is used to store a collection of characters with one character minimally requiring one byte of RAM. Strings may be 'fixed length' (for example, only containing 10 characters), Alternatively, they can use a special 'terminator' character to mark their end. This may mean that a string requires an additional character to be included, but it allows them to have flexible lengths.

Each individual character in a string can be accessed by using its positional index. For example, in Table 4.6, Planet[2] is 'r'.

This example is demonstrated in C# strings in Figure 4.12.

```
string Planet;
Planet = "Earth";

Console.WriteLine("Whole string is {0}", Planet);
Console.WriteLine("3rd character is {0}", Planet[2]);
```

Figure 4.12: Working with C# strings

Table 4.6: Each character in a string has a positional index

Planet					
0	1	**2**	3	4	5
E	a	**r**	t	h	#

Some languages use a static one-dimensional array of characters to simulate a string, rather than using a specific string data type. However, their use is very similar.

Strings are commonly used to store simple text entered by the user, such as a username. Strings may also be used to enter more complex text for processing, such as the content of an email, instant message or tweet. The term 'substring' is used to describe a part of a larger string, for example 'Ear' is a substring of 'Earth'.

Array (one-dimensional)

Traditionally, this is a static data structure, which means that it has a fixed size. However, modern programming languages are generally more flexible and allow an array to be resized.

Typically an array can only store one type of data. Any type of data, such as integers, characters and Booleans, is acceptable.

If we wanted to store seven daily maximum temperatures in degrees Celsius for a local weather station, we would create an array of seven decimal numbers, as shown in Table 4.7.

Table 4.7: One-dimensional array of seven decimal numbers

Temperatures						
0	1	2	3	4	5	6
12.50	10.45	12.30	14.60	17.70	11.20	12.50

This appears to be similar to a string. In fact, you can think of a string as an array of characters. As with a string, it is possible to access individual elements or items in the array by using the required index. For example, Temperatures[4] is 17.70. Figure 4.13 demonstrates the creation and access of this simple one-dimensional array in C#.

Array (two-dimensional)

A two-dimensional array is similar to a one-dimensional array, but it can store multiple rows of data. Imagine that we still wanted to store seven daily maximum temperatures in degrees Celsius for a local weather station, but that we needed to do this over three consecutive weeks. In this case, we would create a two-dimensional array of seven by three decimal numbers, as shown in Table 4.8.

Table 4.8: Two-dimensional array

Temperatures							
	0	**1**	**2**	**3**	**4**	**5**	**6**
0	12.50	10.45	12.30	14.60	17.70	11.20	12.50
1	12.22	11.00	10.00	20.00	21.00	14.00	15.50
2	13.00	14.00	14.50	14.60	12.30	14.00	14.60

As before, it is possible to access individual elements or items in the array by using both indices, for example Temperatures[4][1] is the value found where column 4 and row 1 meet: 21.00.

Tip

The actual order of indices may vary between different programming languages. The general rule is that the element access order will reflect the declaration order of the array, i.e. you could not access an element in Temperature [1][4] as these indices do not reflect the declaration order of the array.

It is possible to use higher orders of dimensions. For example, a three-dimensional array could be used to store the twenty-one temperatures that we have stored for each of five different weather stations.

```
//create the array of decimal temperatures
float[] temperature = new float[7];

//initialise each element
temperature[0] = 12.50f;
temperature[1] = 10.45f;
temperature[2] = 12.30f;
temperature[3] = 14.60f;
temperature[4] = 17.70f;
temperature[5] = 11.20f;
temperature[6] = 12.50f;

//select Thursday's and output it
Console.WriteLine("Temperature on Thursday is {0} degrees C", temperature[4]);

//wait for keypress to continue
Console.ReadLine();
```

Figure 4.13: C# one-dimensional array

Arrays can be split and joined. In some languages, this is called exploding arrays and imploding arrays. In languages where strings are stored as arrays of characters, the extraction of substrings and concatenation of different strings are practical uses of these types of operation. See String handling for more on substring extraction and string concatenation.

Stack (LIFO)

A stack is known as a **LIFO** data structure. It has two basic operations: Push and Pull (sometimes known as Push and Pop).

Stacks are a vital part of any computer platform's operating system, and they are a common tool for a programmer to use when developing solutions. Stacks are conceptually viewed vertically, as in Figure 4.14.

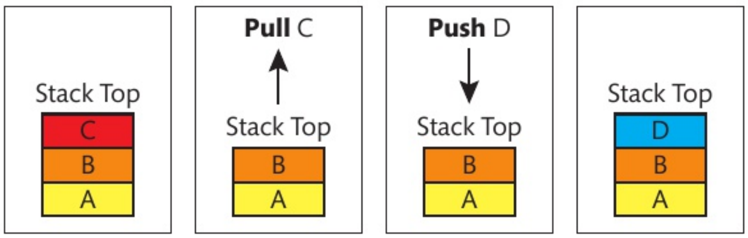

Figure 4.14: Stack operations

You can see a stack operation in C# in Figure 4.15.

```
using System;
using System.Collections;

namespace stack
{
    class Program
    {
        static void Main(string[] args)
        {
            char data;                  //single item of stack data
            Stack st = new Stack();     //our stack

            //push onto stack
            st.Push('A');
            st.Push('B');
            st.Push('C');

            //show the stack
            Console.WriteLine("Current stack: ");
            foreach (char ch in st)
            {
                Console.WriteLine(ch);
            }

            //remove from stack top - it should be "C"
            data = (char)st.Pop();
            Console.WriteLine("The popped value: {0}", data);
```

> **Key term**
>
> **LIFO** – this stands for 'last in, first out' and it describes how data is treated in some data structures. It means that the last item of data pushed on is also the first item of data that may be pulled back off.

```
            //push onto stack again
            st.Push('D');

            //show the changed stack
            Console.WriteLine("Current stack: ");
            foreach (char ch in st)
            {
                Console.WriteLine(ch);
            }

            //wait for keypress to continue
            Console.ReadKey();
        }
    }
}
```

Figure 4.15: Stack operation in C#

Unlike an array, where elements may be accessed in any order, a stack can only be accessed in a last in, first out manner. This can be particularly useful when processing **recursive algorithms**. Stacks are often included in a programming language's library, complete with the necessary methods to process them.

Queue (FIFO)

In contrast to a stack, a simple queue is known as a **FIFO** data structure and has two basic operations:

- add (to tail of queue) or 'enqueue'
- remove (from head of queue) or 'dequeue'.

Only data at the head of the queue can be accessed and removed.

Key terms

Recursive algorithms – a piece of programming code that executes itself repeatedly until it reaches an end condition where the calculated result can be returned.

FIFO – this stands for 'first in, first out'. It means that the first item of data added is also the first item of data that may be removed.

Like stacks, queues are a vital part of any computer platform's operating system. For example, you may be familiar with the concept of a printer queue. For a programmer, they are an excellent way of managing task processing.

Queues are conceptually viewed horizontally as shown in Figure 4.16.

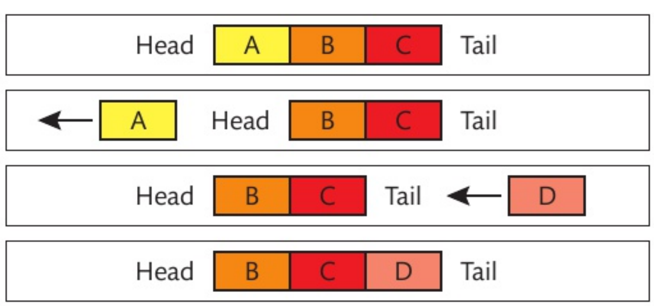

Figure 4.16: Queue operations

Figure 4.17 demonstrates the creation and access of a simple queue in C#.

```
using System;
using System.Collections;
using System.Linq;
using System.Text;

namespace queue
{
    class Program
    {
        static void Main(string[] args)
        {
            char data;                //single item of queue data
            Queue q = new Queue();    //our queue

            //add to queue
            q.Enqueue('A');
            q.Enqueue('B');
            q.Enqueue('C');

            //show the queue
            Console.WriteLine("Current queue: ");
            foreach (char ch in q)
            {
                Console.Write(ch + " ");
            }

            //remove from head - it should be 'A'
            data = (char)q.Dequeue();
            Console.WriteLine("The removed value: {0}", data);

            //add to queue again
            q.Enqueue('D');

            //show the changed queue
            Console.WriteLine("Current queue: ");
            foreach (char ch in q)
            {
                Console.Write(ch + " ");
            }

            //wait for keypress to continue
            Console.ReadKey();
        }
    }
}
```

Figure 4.17: C# simple queue

Record (or structure)

A record is a similar concept to an array. However, it differs because it can store a mix of data types within its structure. For example, it can be used to store details, as shown in Table 4.9.

Table 4.9: A record of student details

Student				
StudentID	**Forename**	**Surname**	**Age**	**Enrolled**
2001	Preston	Myla	21	True

In this example, 'StudentID' and 'Age' are stored as integers, 'Forename' and 'Surname' are strings and 'Enrolled' is a Boolean. A C# implementation of a simple record structure is shown in Figure 4.18.

```
using System;
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{
    public class testStructure
    {
        //create the record structure
        struct Student
        {
            public int StudentID;
            public string Surname;
            public string Forename;
            public int Age;
            public bool Enrolled;
        };

        public static void Main(string[] args)
        {
            Student mystudent;     //declare mystudent of type Student structure

            //initialise the elements in the record structure
            mystudent.StudentID = 2001;
            mystudent.Surname = "Preston";
            mystudent.Forename = "Myla";
            mystudent.Age = 21;
            mystudent.Enrolled = true;

            //print the data in the record structure
            Console.WriteLine("ID         : {0}", mystudent.StudentID);
            Console.WriteLine("Surname    : {0}", mystudent.Surname);
            Console.WriteLine("Firstname : {0}", mystudent.Forename);
            Console.WriteLine("Age        : {0}", mystudent.Age);
            Console.WriteLine("Enrolled   : {0}", mystudent.Enrolled);

            //wait for a keypress to continue
            Console.ReadKey();
        }
    }
}
```

Figure 4.18: C# simple record structure using 'struct'

You can also create an array of structures to store multiple instances of a record. For this reason, a record (or 'struct') can be seen as the basis for simple record storage in a database table or a binary data file.

String handling

Although strings are simply a collection of characters, they may still need to be processed. This task is commonly known by programmers as string handling. Programmers often need to perform certain operations on strings and most high-level languages have specialist library functions to help. Here are some common tasks, with C# practical code examples showing their implementation. Similar string functions exist in many other programming languages.

Finding the length of a string in characters

In Figure 4.19, the length property of the 'month' string object is used to access the string's number of characters. In this example, the string is the word 'January'.

```
string month = "January";
int length;

length = month.Length;
Console.WriteLine("{0} is {1} characters long", month, length);
```

Figure 4.19

Examining single characters

In Figure 4.20, the output will be 'Character 3 of January is u'. This is because 'u' is the character in position 3. The string starts with a 'J' in position 0.

```
string month = "January";
char singleLetter;
int whichLetter = 3;

singleLetter = month[whichLetter];
Console.WriteLine("Character {0} of {1} is {2}", whichLetter, month, singleLetter);
```

Figure 4.20

Extracting a substring

In Figure 4.21, the output will be 'Substring is ua'. This is output because we start extracting at position 3 ('u'), and we want to extract the next 2 characters, including the first one ('ua').

```
string month = "January";
string subString;
int startPos = 3;
int howMany = 2;

subString = month.Substring(startPos, howMany);
Console.WriteLine("Substring is {0}", subString);
```

Figure 4.21

Concatenating two strings

Concatenation is the process of joining things together. In programming, it is used to describe the process of joining shorter strings together to form longer ones, for example 'Hello' and 'World!' to form 'Hello World!'.

In Figure 4.22, we are joining two strings: 'January' and 'has 31 days'. The output will be 'Joined string is January has 31 days'. This is output because we use the '+' operator to join the two strings together. Other techniques are available in C#, but this is perhaps the simplest to use.

```
string month = "January";
string saying = " has 31 days.";
string combined;

combined = month + saying;
Console.WriteLine("Joined string is {0}", combined);
```

Figure 4.22

File handling

Data storage is a key aspect of modern programming. The ability to open data files for reading (i.e. getting data from a file) and writing (i.e. sending data to a file) in a programming language is essential. In order to use a file, it must be opened, and when you have finished it should be closed. Some programming languages require special commands to perform these operations. For example, C uses fopen() and fclose().

A simple form of file handling is the ability to read data from an ASCII or text file. This can be achieved in Microsoft C# using a few simple constructs, as shown in Figure 4.23.

```
using System;
using System.IO;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            //exception handling
            try
            {
                // attempt to read the named file
                using (StreamReader file = new StreamReader("c:/customer.txt"))
                {
                    string line; //stores a single line of text read from the file

                    //loop which continues to read a line of text until
                    //end of file (EOF) is reached
                    while ((line = file.ReadLine()) != null)
                    {
                        //output the line of text read
                        Console.WriteLine(line);
                    }
                }
            }
            catch (Exception e)
```

```
                {
                    //show user the correct error message.
                    Console.WriteLine("Sorry, could not read the file.");
                    Console.WriteLine(e.Message);
                }
                Console.ReadKey();
            }
        }
}
```

Figure 4.23: C# file handling demonstrating the reading of an ASCII/text file

In Figure 4.23, a simple C# 'while' loop is used to iterate through each line of the specified text file. As the line of text is read, it is output to the screen. Exception handling is used to ensure that the program does not fatally crash if the named text file cannot be found.

Most modern programming languages (such as Microsoft C#, PHP, Ruby and so on) also support connection to relational databases such as Microsoft SQL and MySQL. This supports more complex data handling.

Event handling

Event handling describes the complex process of using a specially written function or class method to perform set actions when a user event or system event is triggered. Examples of this include outputting the result of a calculation when a button is pressed on a screen form, or displaying a confirmation message when the screen resolution is changed.

Events are more commonly found in form-based applications, such as those which use **graphical user interface (GUI)** elements to allow the user to interact with the application. It is possible to create simple event-handling applications that work in the console or **command line interface (CLI)** environment, as shown in Figure 4.24.

Key terms

Graphical user interface (GUI) – a modern user interface comprised of windows and icons which is controlled by a mouse and a cursor/pointer. Examples include Microsoft Windows 10 or Apple OS X.

Command line interface (CLI) – an older style of text-based user interface that is still in use. Users interact with the computer via commands entered from the keyboard. Examples include Microsoft Command Prompt or a Linux terminal.

```
//Publisher - defines the event and the delegate
public class myClock
{
    //Defines the delegate, needed to define an event inside a class
    public delegate void SecondHandler(myClock clock, EventArgs evt);
    //Defines an event based on the delegate
    public event SecondHandler second;
    public EventArgs evt = null;

    public void Start()
    {
        //never stop!
        while (true)
        {
            //1000 milliseconds is 1 second
            System.Threading.Thread.Sleep(1000);
            if (second != null)
            {
                //raise the event (each second)
                second(this, evt);
            }
```

```
        }
    }
}

//Subscriber - accepts the event and provides the event handler (showClock)
public class Listener
{
    public void Subscribe(myClock clock)
    {
        clock.second += new myClock.SecondHandler(showClock);
    }

    //Event Handler - shows the clock in the top-left corner of the screen
    private void showClock(myClock clock, EventArgs evt)
    {
        Console.CursorLeft = 1;
        Console.CursorTop = 1;
        DateTime now = DateTime.Now;
        Console.WriteLine(now);
    }
}
//class used to test the event-driven clock
class Test
{
    static void Main()
    {
        //create a new clock object
        myClock clock = new myClock();
        //create a new listener object
        Listener myEventListener = new Listener();
        //link the event and the handler
        myEventListener.Subscribe(clock);
        //start the clock ticking...
        clock.Start();
    }
}
```

Figure 4.26: Using C# console events to create a real-time clock

In this example, a one-second event is used to provide a real-time clock on the console screen. This is an example of C#'s Publisher-Subscriber model. The Publisher defines the event and the delegate. Once the Publisher invokes or triggers the event, other objects in the program such as the Subscriber are notified. The Subscriber then accepts the event and provides an appropriate event handler. In this case, the appropriate event handler is a function which displays the clock in the top-left corner of the screen.

Link

Form-based event handling plays a significant role in developing mobile apps, as demonstrated in *Unit 7: Mobile Apps Development*.

Documentation of code

Program code should always be documented to good standards. Doing this will:

- make the code more readable
- help other programmers understand the code
- help when programmers are debugging the code
- aid with maintenance of the program code, especially if this will be performed by another developer.

Programmers can use various techniques to document their code. These techniques include:

- self-documenting identifiers – this means using meaningful names which explain the purpose (and sometimes the data type) of the value represented by the identifier

Tip

Rules on creating identifier names can vary greatly between programming languages. Some identifiers can only use certain characters (e.g. no spaces) or cannot start with certain characters (e.g. digits 0–9). In addition, stylistic advice is often given, e.g. do not abbreviate or use acronyms when naming identifiers.

- single-line comments – these are short comments that explain the purpose of a line of code
- multi-line comments – these are longer comments that explain the purpose of a more complex section of code or preface a separate section of code, such as a function, procedure or class
- use of constants – this means replacing multiple instances of fixed 'literal' values (numeric, text etc.) scattered throughout a program with named identifiers which may be modified at a single point in the program code.

Table 4.10 shows these most common techniques in C#.

Table 4.10: Documentation of code in Microsoft C#

Documentation concept	Microsoft C#
Self-documenting identifiers	`int userAge;`
Single-line comments	`// stores user's age`

Multi-line comments	`/*`
	`This section of code validates the user's age when entered.`
	`*/`

Use of constants	`const double PI = 3.14159;`

Research

Many software publishers recommend standards for naming identifiers. For example, Microsoft provides online documentation containing advice and guidance on these technical issues in order to encourage good programming practices. Visit their Developer Network to find out about their recommended identifier naming conventions.

Link

Tools are also available which generate HTML documentation directly from comments placed in programming code. This is discussed further in *Unit 7: Mobile Apps Development*.

Principles of logic applied to program design

In computing, logic is the set of principles beneath the different elements of a program or system that allows the program or system to function. Successful program design benefits from the application of simple logic. This is more important than the ability to think in a particular programming language. Being able to think logically will help you to process data correctly, form complex conditions which enable you to test data appropriately and build powerful algorithms to solve even the most complex problems.

The principles of logic used in computing include mathematical logic, iterations, propositional logic and the use of sets. These concepts help programmers to understand problems more fully and build the logic that is incorporated into their programs.

Mathematical logic

Mathematical logic can be applied to help developers understand and accurately describe the basic facts in a problem and their relationships to one another. For example, inference is the process used to reach a logical conclusion from premises (situations) which are thought or known to be true. This can be seen in the following example.

- Premise: All meat comes from animals. This is true.
- Premise: Lamb comes from sheep. This is true.
- Inference: Therefore lamb comes from animals.

This inference is true because sheep are animals. False inferences are called fallacies.

The concept and use of inference is important when using programming languages such as Prolog ('Programming in Logic') to build simple knowledge bases which can be queried to reveal new facts.

Other logical concepts can also be considered. For example, consistency tries to see if a statement (and its opposite) are both true in order to determine whether there are contradictions in the statement.

Iteration

Iteration occurs when a computational procedure is applied to the result of a previous application.

Completeness

Completeness is a logical concept that states that you can 'prove' anything that is correct. Here is a simple assertion which uses the following symbols:

- Σ (the Greek letter sigma)
- Φ (the Greek letter phi)
- $\vdash$ (turnstile) means 'proves' or 'yields'.

Writing the assertion '$\Sigma \vdash \Phi$' means 'from Σ, I know that Φ'. More simply, it means that Σ 'proves' Φ.

There are many different forms of completeness, such as functional and semantic. In programming, completeness can be used to prove whether algorithms would successfully work given certain rules, goals and logic.

Truth tables

You can verify simple algorithmic logic through the use of truth tables. For example, a customer can only log in (L) to a system if they have a valid username (U) and password (P), and have not been locked out (O) within the last 30 minutes.

In a truth table for this example, 1 is used to represent 'true', while '0' means false. The '.' symbol means logical 'and'. The bar above the 'O' means logical 'not'. This means that we can express this logical expression as L = (U.P). $\overline{O}$ or 'successful login = username and password and not locked out'. After the expression is evaluated, we can prove the only combination which is valid, as shown in Table 4.11. The resulting expression could be converted to any target programming language, as its underlying logic is proven by the truth table.

Table 4.11: A truth table showing possible combinations and the only valid permutation (row in bold)

U	P	O	L (U.P). $\overline{O}$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	**1**	**0**	**1**
1	1	1	0

Propositional logic

Propositional logic is based on the simple concept that a sentence can be considered to be either true or false, but not both true and false. Consider these examples.

- The Sun is hot.
- Two plus three is six.

Both sentences are propositions and they have a truth value of either 'true' or 'false'. The first example is true because the Sun is indeed hot. The second example is a false proposition, because two plus three is five, not six. These types of simple sentences are known as basic propositions.

More complex propositions can be created using connectives. There are five basic connectives. Each one has a word and symbol form, as shown in the photos.

$\neg$

NOT

$\wedge$

AND

$\vee$

OR

$\rightarrow$

IF_THEN (or Imply)

$\leftrightarrow$

IF_AND_ONLY_IF

These basic connective symbols allow us to represent complex expressions in a very mathematical fashion.

For example, the expression, 'If I have **m**oney **or** a **t**icket then I can watch a **f**ilm' could be expressed as: '$(M \vee T) \rightarrow F$'. In other words: F is true if M or T is true.

In another example, the expression, '**If** I have **no m**oney, then **if d**irect **d**ebit is due, a **b**ank **f**ine must occur' could be expressed as: '$\neg M \rightarrow (DD \rightarrow BF)$'. In other words: if M is false, then if DD is true, BF must be true.

Because such simple logic statements use a formal mathematical notation, it is possible to test their validity before any program code is produced. It is also possible to use propositional logic to derive new knowledge from existing facts.

A further revision called propositional dynamic logic (PDL) is used to demonstrate the function of algorithms. PDL is designed to present the states and the events of dynamic systems (such as working programs) in mathematically friendly notation. One of its most important features is its ability to check whether completed algorithms can complete with the desired state: that is, calculate the correct result.

Use of sets

Sets can be used to help organise data and define the interrelationships between different sets of data. This allows you to easily search and **filter** potentially complex data. Some programming languages have data types which support set-style functionality, and this logic can be used to form user permissions.

Key terms

Set – a collection of distinct objects. Sets can contain anything (e.g. names, numbers, colours or letters of the alphabet) and may consist of many different members.

Filter – include or exclude certain values when running a search.

For example, you may have two groups of people with different access rights in a particular program. Set A is the Admin group, which contains John, Ahmed and Jo. This can be expressed as Set A = {John, Ahmed, Jo}. Set B is the Finance group, which contains Claire, Jo, Niamh and Phil. This can be expressed as Set B = {Claire, Jo, Niamh, Phil}. We can represent these sets as a Venn diagram, as shown in Figure 4.25.

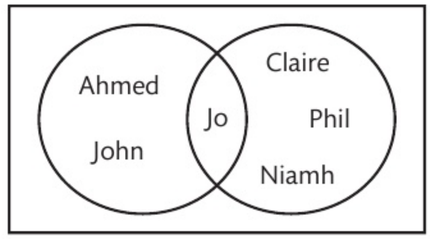

Figure 4.25: Venn diagram showing two sets of people

There is a particular function in the program that can only be accessed by administrators (members of the Admin group) who work in the Finance department (members of the Finance Group). Looking at Figure 4.25, we can easily see that the only user who exists in both sets is Jo. This means that only Jo has access to this function.

Quality of software applications

When faced with the same problem, different programmers can arrive at very different solutions. This usually occurs because problem solving is a very personal process and most programming languages have enough flexibility to permit a variety of solutions to the same problem. However, the actual design and implementation of the software solution has a direct impact on its overall quality. The quality of a software solution is judged by assessing its efficiency or performance, maintainability, portability, reliability, robustness and usability.

Discussion

In small groups, share your experiences of having problems with unreliable software applications. Do you think that these problems could have been prevented? Why do you think these problems were not fixed before the application was released?

Efficiency or performance

The performance of an application is usually assessed by measuring the system resources consumed by the program. For example, CPU clock cycles, processor time, allocated RAM and the rate at which storage media is read from and written to. A complex process may be very

CPU intensive. It may be possible to revise the program code to be more economical, which, in turn, may improve performance (work faster) and use fewer resources (RAM and so on). This type of revision is common in programs which rely on speed, such as videogame programming (where screen frame rates can suffer due to inefficient coding) or talking to external devices in real-time systems where timing issues can be critical, for example traffic control systems or missile guidance systems.

Maintainability

Maintainability is the ease with which a program can be modified by present or future developers in order to carry out **corrective**, **perfective** or **adaptive maintenance**. For example, code that is written in a maintainable way will be easier to adapt or modify, and this will save both development time and money. There is always a trade-off to be made between writing code efficiently to reduce initial development costs and considering how long future maintenance will take.

Key terms

Corrective maintenance – fixing an error or bug that has been identified.

Perfective maintenance – making an improvement to a program that enhances its performance

Adaptive maintenance – making modifications to a program, by adding, changing or removing functionality to reflect changing needs.

Portability

Portability is a measure of the number of different computer platforms that the source code can target, such as hardware, operating systems and so on. Some programming languages, such as C, are particularly portable, as they have compilers that translate the existing code to the required target machine code on many different platforms. For example, a program written using Microsoft Visual Basic .NET may limit its execution to a Microsoft operating system, while programs written in C can be ported for use in Microsoft or Linux operating systems with few platform-specific code changes.

Reliability

The reliability of an application is determined by the overall accuracy and the consistency of its outputs across multiple runs. Consistency is particularly important in reliability testing. A program may calculate answers very accurately, but, if it cannot replicate this every single time it runs, it is not considered to be reliable. This is even more important in environments where people's safety depends on the reliability of a program, such as in safety systems in a power station or collision detection in aircraft traffic control systems.

Discussion

We have already seen a couple of examples where reliability is crucial. In pairs, discuss and write down three more examples of situations in which a program has to be completely reliable.

Robustness

Robustness is a measure of the program's code quality, particularly when the program is tested to ensure that extreme and erroneous data can be processed without causing the program to crash.

Usability

Usability is a measure of how user-friendly an application is to use. If the principles of user-centred design (UCD) have been followed, users will have been involved throughout the design process. Their ongoing evaluation will have helped to refine the design many times before being fully accepted. Popular techniques, such as rapid application development (RAD), permit the developer to quickly alter the layout, flow and interactivity schemes of an application to receive realistic interface feedback from the user. Although this form of rapid prototyping usually lacks the full functionality of the final product, it does provide the targeted user with a fairly-realistic 'look and feel' of the application quite early in the development process.

Usability includes a program's inputs, navigation and standards of output. The general appearance of the application can also have a bearing on this. For example, usability may suffer if the choice of colours or fonts is poor. Accessibility for users with disabilities or special needs should also be considered when thinking about usability, particularly for users with physical, visual or hearing impairments who may use **assistive technologies**. The broader study of human-computer interaction (HCI) can also inform the design of application interfaces, including how the user's psychology and, in particular, their behaviour, affects their use of the application.

Key term

Assistive technologies – hardware or software designed to assist users with a specific disability or special need, such as screen readers for users with visual impairments or learning disabilities.

PAUSE POINT Can you explain what the learning aim was about? What elements did you find easiest?

Hint Close the book and make a list of the features of computational thinking skills and the principles of computer programming.

Extend How does the type of programming language affect the development of a software solution?

Assessment practice 4.1

A.P1 A.P2 A.P3 A.M1 A.D1

A local school is launching a campaign to promote programming. You have been asked to write a student-friendly blog post which explains and promotes the application of computational thinking skills when solving problems and developing software applications.

To make the blog post as helpful as possible for your target audience, you should explain the principles of computer programming, especially how different languages can be used to create solutions. You should also discuss the basic principles of software design and how they are applied to produce the high-quality software that your target audience needs and uses on a daily basis.

Conclude your blog post by analysing and evaluating the impact of computational thinking on software design and the quality of the software produced.

While you are writing your blog post, think carefully about your target audience and their level of knowledge about programming. Consider how you will explain technical concepts to an audience that may contain both experts and beginners.

Plan

- What is the task? What am I being asked to do?
- How confident do I feel in my own ability to complete this task?
- Are there any areas I think I may struggle with?
- Am I using appropriate language for the target audience?
- Do I understand the difference between analysis and evaluation?

Do

- I know what I am doing and what I want to achieve.
- I can identify where I have gone wrong and adjust my thinking or approach to get myself back on course.

Review

- I can explain what the task was and how I approached it.
- I can explain how I would approach the more difficult elements differently next time (i.e. what I would do differently).

B Design a software solution to meet client requirements

The key point of any software solution is that it meets the client's requirements. If you do not or cannot achieve this, then it does not matter how attractive, user-friendly or efficient your solution is: you have not actually solved your client's problem.

Make sure that you really understand your client's requirements before you start. Once you have started to design and develop the solution, you should also return to the original client requirements at regular intervals. This will help you to keep your software development on course for success.

Software development life cycle

Designing a software solution is a cyclical process with clearly defined stages. Although many different versions of the software development life cycle exist and names may change, most stages are common and follow the same sensible order.

The software development life cycle (SDLC) is a **conceptual model**. It describes the stages used to manage the creation of a software solution from its inception through to its ongoing maintenance and eventual retirement or replacement. A typical SDLC is shown in Figure 4.26.

Key term

Conceptual model – a way of organising ideas and concepts in a logical fashion. Conceptual models often represent the ideas concerned in a visual manner that illustrates the relationships between them in a simple way that is easily understood.

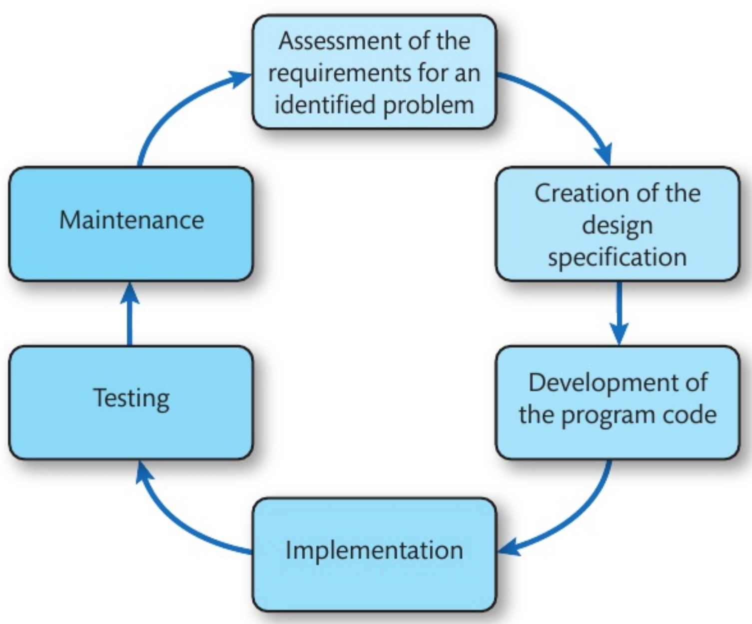

Figure 4.26: SDLC conceptual model - there is a big overlap between development of the program code and implementation

The following step-by-step process shows how these six steps are applied in software development.

Step by step: Software Development Life Cycle (SDLC)

6 Steps

1 Assessment of the requirements for an identified problem

You must fully understand the client's requirements for an identified problem. If you do not have a clear understanding of what needs to be achieved, it will be difficult or even impossible to create a working solution. The requirements for a solution to the identified problem are usually drawn from the client's brief. Where details are lacking or clarification is needed, you must investigate in order to grasp what the client wants the program to achieve. This may involve a number of different investigation techniques including:

- direct questioning of users or employees about the tasks they perform, their work patterns and any difficulties that they encounter in their jobs
- observation of the manual process which is being computerised, such as completing a customer enquiry form or calculating the cost of an order
- examination of manual documents which need to be computerised. This might include data collection forms such as order forms, timesheets or stock lists, and outputs such as reports or charts.

Once the requirements are clearly set out, it is possible to decide what is within the scope of the project and what is not within the scope. From this, you can create the design specification.

2 Creation of the design specification

A full design specification needs to include the following items.

- The scope of the project: This is what is covered by your proposed solution and it may also be called the 'problem domain'. Defining a project's scope upfront helps a software developer to stay on track and avoid 'feature creep' – this means adding features which are not initially required, which will potentially delay completion of the solution.
- Inputs: These are the values being entered into the program and how these values will be entered (e.g. manually, automatically from a database).
- Outputs: This is the information generated by the program. You should include details about its format and layout.
- User interface: This is how the user will interact with the program (e.g. menu systems, use of keyboard and mouse, voice control, colour schemes and layout).
- Processes and algorithms: These include calculations being performed in order to generate the desired output from the data being entered, data being validated and so on.
- Timescale: This is how long the process will take, including targets to be completed during the duration of the project. It should also include agreed 'milestones', which are predetermined points in the process at which the developer can demonstrate how much progress has been made.

3 Development of the program code

Program code should be written in your chosen language, although sometimes a client will require a specific language. The choice will be based on the design specification and the specific requirements of the solution. As you have already seen, there are many different factors that will influence your choice of programming language. The documented inputs, outputs, processes, algorithms and user interface required should provide a software developer with a very clear vision of what the program is and how it should work. The developer has to take the design specification and use their chosen programming language as efficiently as possible to create working program code, using the most appropriate features of the selected language.

In a commercial environment, this is typically achieved using discrete steps, known as 'story points'. These break up the different parts of the program being worked on into hours of work, with one hour typically equating to one story point. This means that more complex features will equate to more story points.

Commercial software developers generally produce program code on a 60/40 basis. This means that they spend 60 per cent of their time coding and 40 per cent planning, debugging and testing.

4 Implementation

Implementation involves:

- selecting the most appropriate programming language (if this has not already been chosen by the client)
- selecting the development environment – the software tool(s) used to build the program, the operating system and sometimes the type of hardware
- coding the solution from the design specification
- debugging the code to ensure that simple bugs are identified and removed before formal testing starts.

5 Testing

Testing is an essential process that ensures that any program fully meets the client's requirements and operates in an accurate, reliable and robust manner.

There are two common types of testing: white box testing and black box testing.

- White box testing is usually performed by the developer who has produced the program. It involves tracing the **use cases** through the program code logic and completing **trace tables**.
- Black box testing is performed by a user following a use case. The user has no exposure to the program code and does not need to know how the program works. You, as the developer, are only interested in the outcome the user gets.

Testing enables the program to be refined (made more precise or exact) and optimised (made to work faster or more efficiently).

Key terms

Use case – a list of specific actions or events which occur between a user and the program. Possible use cases that occur when a customer tries to withdraw cash at an ATM include 'card rejected', 'PIN correct', 'PIN incorrect', 'card swallowed', 'cash dispensed' and 'no cash available'.

Trace table – a table that tracks inputs, processes and outputs for each use case. It includes an expected result (what should happen) and an actual result (the result of the program), which can be compared and contrasted to identify unexpected outcomes.

6 Maintenance

Maintenance is an ongoing process. It results in correcting the program code based on testing and/or user feedback.

- Maintenance may adapt the original solution to meet changing client needs. For example, if a client originally required financial information in one currency and now requires it in multiple currencies instead, this would be an adaptation of existing code.

Alternatively, maintenance may expand the solution by including additional functionality. For example, if a client asks for completely new menu options to fulfil a totally new requirement, this would be an expansion to include additional functionality.

The whole process of the software development lifecycle may be repeated over and over again. The performance of any programmed solution will eventually decline over time as the client's needs or business circumstances change. Active maintenance can prolong a program's life, but, eventually, the cycle will restart and requirements for a new improved solution will need to be assessed.

Software solutions design

A software solutions design or specification is a formal document that is completed by the developers (with input from the clients and/or target users) before the actual programming starts. It should include a full breakdown of the problem-solving process

that has been followed and all the suggested solutions. If written correctly, it should provide enough detail for a programmer who is unfamiliar with the problem to build the desired application using the preferred solution.

Tip

Remember that there may be more than one solution to consider for a single problem.

Problem definition statements

Problem definition statements are a key element of software solutions design. They are clear descriptions of the issues that exist, the people affected by the issues and the **constraints** which may have an impact on the solution. Typically, they are created during the initial investigation and discussion with clients by asking the five 'Ws': who, what, when, where and why? They help the developers to understand the problem, its scope and its constraints. In this way they help to focus the minds of the developers when they are solving the problem.

'What?', 'When?' and 'Where?' questions will help you to create a full summary of the problem to be solved and its complexity, as well as the benefits of and constraints on any proposed solutions.

- What is the **context** of this problem?
- What is the nature of the problem to be solved?
- What are the boundaries or scope of the problem?
- What requirements are defined by the client?
- What are the constraints that limit the solution that can be developed?
- What are the benefits of solving this problem?
- What is the impact of not solving this problem?
- What is the purpose of the required solution?
- What is the complexity of the problem we need to solve?
- What is the nature of the interactivity between the target users and the solution?
- What level of cooperation will the software designers receive from existing users and clients when solving the problem?
- What resources are available to solve this problem?
- When is the problem occurring? Can its timing be isolated? Is there a recognisable pattern?
- When does this solution need to be operational?
- Where is the problem occurring? Can its location be isolated? Is there a recognisable pattern?
- Where is the solution to be deployed?

'Who?' questions will help to identify the intended users of a software solution and the nature of the interaction that users will have with the software solution.

- Who is affected by this problem?
- Who wants this problem to be solved?

'Why?' questions will help to further identify the benefits of a software solution.

- Why does this problem occur? Can its cause be isolated? Is there a recognisable pattern?
- Why does this problem need to be solved?

Key terms

Constraints – restrictions on something. Constraints in a programming context include features of the programming languages, the technical skills of the developer, the platforms supported by the programming language and so on.

Context – the setting or circumstances surrounding something. In software solutions design, the context will include details such as the background history of the problem.

Worked example: Writing a problem definition statement using a 'What?' question

A builders' merchant has a problem with the integration of their store and online systems. The developer asks the client, 'What is the nature of the problem to be solved?'

This is the problem definition statement that the developer creates based on the client's answers to this 'What?' question:

'A customer may have two different trade accounts which they can use within a branch store to buy goods on credit. However, when they visit our online store, they can only use the new type of account – older account types simply are not recognised and we have received complaints. We need to allow the customer to select which account they want to use to purchase our goods by choosing the correct account from an available list on screen.'

This problem definition statement is clear and concise. It describes the exact problem that needs to be solved using the correct terminology for the client's sector.

Theory into practice

Using the worked example, write problem definition statements that answer these questions.

- What are the benefits of solving this problem?
- What is the impact of not solving this problem?

Features of software

A description of the features of the proposed software solution is a core part of the solution design. This includes the main program tasks, any data storage that is required and the required inputs and outputs.

Main program tasks and input and output formats

When considering the required input and output values, you should also think about the formats that are required. For example, a developer has been briefed to create a simple program to convert an amount of pounds sterling (£) into US dollars ($) or euros (€) depending on the user's choice. The developer has created a visual representation of the required inputs, processes, outputs and data storage, as shown in Figure 4.27.

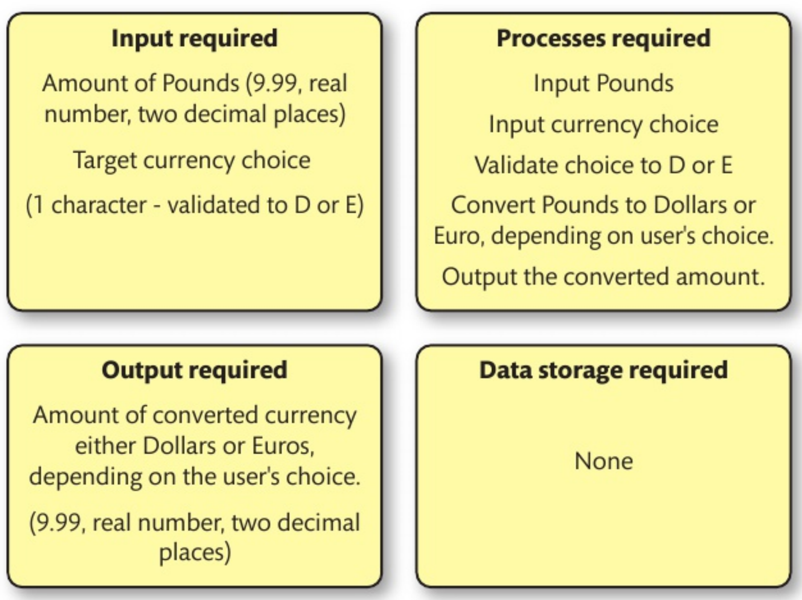

Figure 4.27: A quad diagram showing inputs, outputs and processes

The developer has listed all the inputs and outputs and noted their formats, particularly the quantity of decimal places to use. They have also included details of any validation that is required.

Tip

When specifying formats, it is typical to use 9 for a numeric digit, A for an alphabetic character and X for any character.

Diagrammatic illustrations

A solutions design will almost certainly contain a number of different diagrams, or diagrammatic illustrations. There are three common types that can be included, each of which focuses on aspects of the **user experience (UX)**. These are:

- screen layouts, showing how elements will be organised on the virtual 'page'
- user interfaces, showing how the user will interact with the application
- navigation elements, showing how the user will move between different virtual 'pages'.

These illustrations (see Figure 4.28) can be sketched out on paper or created electronically using specialised design tools, many of which are available online.

Key term

User experience (UX) – a measure of how the user interacts with the program and their satisfaction when using it.

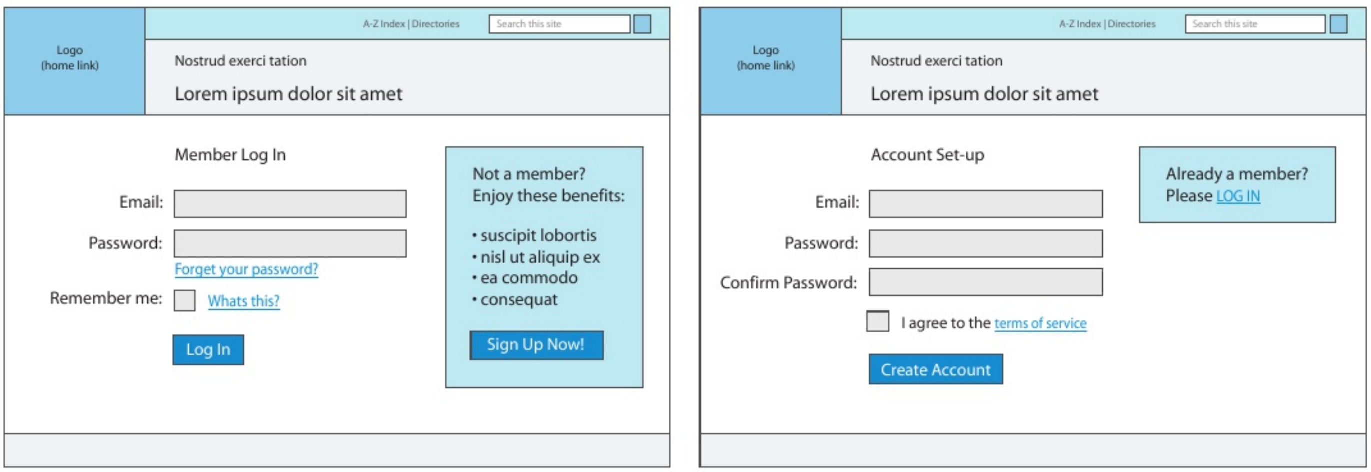

Figure 4.28: Two potential screen layouts for the same application

Key term

Iteration – a repetitive process, usually something done repeatedly until it is correct.

Table 4.12 demonstrates the advantages and disadvantages of diagrammatic illustrations.

Table 4.12: The advantages and disadvantages of diagrammatic illustrations

Advantages	Disadvantages
The creation process can be sped up when illustrations are created using electronic design tools.	Electronic design tools may require registration or purchase.
Can be easy to create and quick to change, especially if created electronically.	Can be time consuming if they have to be **iterated** through a number of design revisions.
Allow the designer to walk potential users through the flow of the application. This may help to identify any areas of confusion or difficulty while the design is being refined.	Can be difficult to replicate on-screen appearance exactly.
Allow the client to view prototypes of the intended user experience before the coding begins.	
Improve client involvement by helping to involve clients in the design process. This means that the final application has the best possible chance of fully meeting their requirements.	
Allow designers to try different ideas and get feedback on their ideas.	
Save development time, as developers do not have to repeatedly adjust the program code, especially as adjustments may introduce bugs.	

> **Link**
>
> Screen layouts and navigation for mobile apps (a technique called wireframing) is discussed in *Unit 7: Mobile Apps Development*.

Algorithms

An algorithm is a set of instructions that is followed to solve a problem or perform a particular stage of processing in the overall solution, such as validating user inputs. Software applications may be made up of many different algorithms, implemented in the program code using a combination of many different programming constructs, functions and procedures.

> **Link**
>
> Algorithms used to design mobile apps are discussed in *Unit 7: Mobile Apps Development*.

Because they can be difficult to communicate, algorithms may be represented using a number of different design tools, some which offer diagrammatic illustration.

The three most common tools are pseudocode, flowcharts and event diagrams or tables.

Pseudocode

Pseudocode is an informal outline of the algorithm, expressed in natural language. It can be converted into the target programming language. For example Figure 4.29 shows a simple validation of a user's age between 18 and 60. Pseudocode should not contain any programming language commands or syntax.

```
Do
   Ask user for their age

   Input user's age

   If age is less than 18 or
   greater than 60 then

      Output age error

   Else

      Output age accepted

   Endif

While age is not between 18 and 60.
```

Figure 4.29: Simple validation of user's age

Flowcharts

A flowchart is a graphical representation of the algorithm, showing its actions and logic through a set of standardised symbols. Figure 4.30 shows the same algorithm as in Figure 4.29, but represented as an extract from a flowchart.

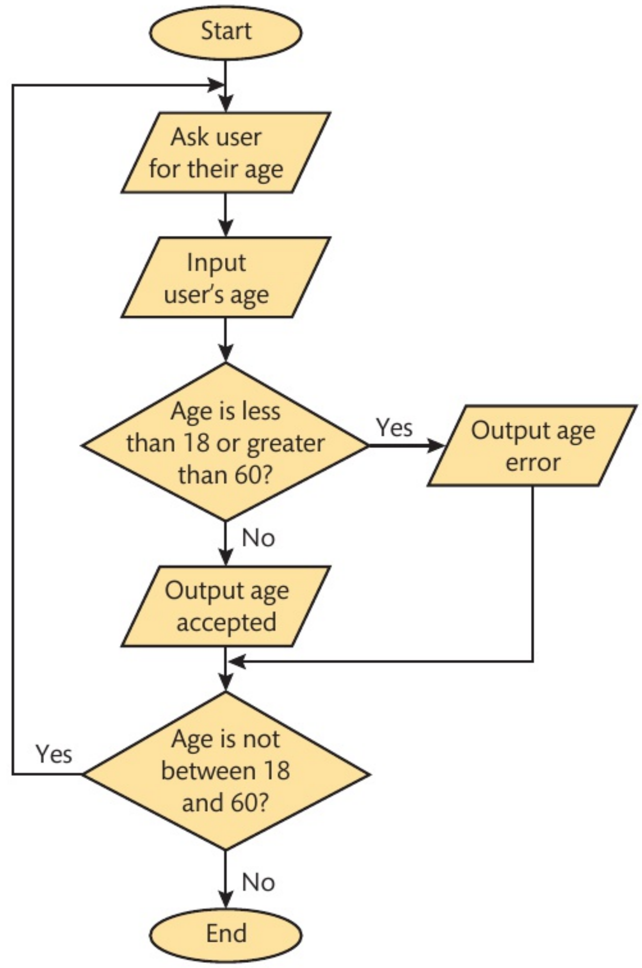

Figure 4.30: User's age expressed as a flowchart

Event diagrams

Event diagrams or event tables are used to record events in a simple table. For each event, the following details are included in the table.

- The name of the container in which the responding object is located (e.g. Form1).
- The name of the object responding to the event (e.g. Button1).
- The event being responded to (e.g. Click).
- The event handler which is triggered by the listener (e.g. button1Action()).

Event tables can act as a useful checklist when building the GUI components of an application.

> **Research**
>
> Although flowcharts can be created manually, many free online tools exist that support their creation in a user-friendly 'drag-and-drop' fashion.

Data structures

Any software solution should detail the different structures which may be used as part of any algorithm. These are different from data storage as data structures are stored in RAM while the application is executing.

Common data structures include arrays (one-, two- and three-dimensional), queues, stacks and records.

Data storage

Designing any software solution involves dealing with the persistence or continued existence of data. A computer's data storage in RAM is said to be 'volatile' or unstable because it is lost when power is removed, for example when the computer is switched off.

This means that, while it is fine to store data in RAM while the program is running, you need a more permanent (non-volatile) form of data storage to enable data to exist between program uses, especially when the power is removed. Non-volatile forms of data storage include magnetic storage, such as a hard disk, or a USB flash drive.

The most common form of non-volatile data storage is the data file. These are supported by most programming languages. Although the data file stores the data used by a program on non-volatile media, it does not store the instructions of the program itself. These are kept in separate files.

Data files may exist in many different formats. Common examples include:

- ASCII (American Standard Code for Information Interchange) or text
- Binary(where numbers are stored as a pure binary value not individual ASCII character codes)
- CSV (Comma Separated Value)
- XML (eXtensible Markup Language).

Table 4.13 demonstrates how the same customer data can be specified in in ASCII, CSV and XML for comparison.

The choice of the data file format is often influenced by its use. For example, if a data file is to be imported into a spreadsheet application (such as Microsoft Excel), CSV is a popular option. Data files can be used for many purposes, such as storing user records, a program's configuration information or its licence details.

Designing a software solution requires you to consider data storage in the following ways.

- What data storage is required to solve the problem (e.g. customer data and product data)?
- What should the files be called?
- How will the data be accessed (e.g. read, write or **append**)?
- If the data will be updated, deleted and so on, how will this happen?
- What file format will be used (e.g. ASCII, CSV or XML)?
- Where should the file be stored (e.g. media and folder name)?

Key term

Append – to add to the end of something. In a data file, it means adding new data to the end.

Table 4.13: Comparison of different file formats used for simple data storage

Data file format	Data storage example	Notes

ASCII	`CustomerAcc Lastname Firstname Location`
	`000000001 Jones Alex London`
	`000000003 Willis Claire Swindon`
		The first row is used for headings. A tab is used to separate each value. A new line separates each customer's data.

CSV	`CustomerAcc,Lastname,Firstname,Location`
	`000000001,Jones,Alex,London`
	`000000003,Willis,Claire,Swindon`
		The first row is used for headings and each value is separated by a comma. A new line separates each customer's data.

XML	`<?xml version="1.0"?>`
	`<Customer>`
	`<CustomerAcc>000000001</CustomerAcc>`
	`<Lastname>Jones</Lastname>`
	`<Firstname>Alex</Firstname>`
	`<Location>London</Location>`
	`</Customer>`
	`<Customer>`
	`<CustomerAcc>000000003</CustomerAcc>`
	`<Lastname>Willis</Lastname>`
	`<Firstname>Claire</Firstname>`
	`<Location>Swindon</Location>`
	`</Customer>`
		The XML document starts with an XML tag stating its version. Each customer is 'blocked' or separated from others using a Customer start-tag and end-tag pair. Each piece of data is placed in a start- and end-tag pair describing the attribute, e.g. CustomerAcc.

More complex solutions may require many data files in order to function correctly. In some cases, particularly when searching functionality and when complex data relationships are needed, it may be more appropriate to make use of a relational database instead of data files.

Control structures

Control structures include sequences, selections and iterations. The validation algorithm shown in Figures 4.29 and 4.30 contains all three control structures. A software solution must clearly show the range of control structures selected and how they are used to solve complex problems as part of an algorithm. Flowcharts and pseudocode are good techniques to use for highlighting these details.

Data validation

You may have heard the expression 'garbage in, garbage out' (GIGO). This describes the general rule that the quality of the output is directly dependent on the quality of the input. If the input is rubbish, the output will also be rubbish. Validation is the process that checks to see if an inputted value makes sense and is reasonable before it is processed.

The use of checkboxes, buttons and list boxes usually limits input choices to sensible inputs. However, most programs still require validation to handle the probability of problematic inputs from the user, particularly when using traditional keyboard inputs.

Tip

Validation does not check to see if the data has been keyed in accurately. This is a separate process called verification.

It is very important to build validation rules into a solution. They will check whether different inputs are sensible and prevent inaccurate results, **run-time errors** or fatal application crashes.

Key term

Run-time error – a problem that occurs while an application is being used. These errors result in the application locking (refusing to accept user input) or crashing (terminating and returning the user to the device's menu or desktop).

There are several different types of validation.

Range check

A range check assesses whether data entered is within a valid minimum-to-maximum range. For example, if a customer is limited to buying up to ten of a particular item, then the valid range would be 0 to 10. Input outside this inclusive range would be considered invalid. Typical C# program code to perform this type of validation is shown in Figure 4.31.

```
const int MIN = 0;      //minimum value of the range
const int MAX = 10;     //maximum value of the range

string strNumber;       //string to temporarily store our input
int qtyValue;           //our inputted quantity

//perform loop while quantity is outside range
do
{
    //input quantity
    Console.WriteLine("Enter quantity between {0} and {1}:", MIN, MAX);
    strNumber = Console.ReadLine();
    qtyValue = int.Parse(strNumber);

    //check is quantity outside range
    if (qtyValue < MIN || qtyValue > MAX)
    {
        Console.WriteLine("Sorry, the quantity entered is outside the allowed range");
    }

} while (qtyValue < MIN || qtyValue > MAX);

Console.WriteLine("Your quantity of {0} is valid, thank you.",qtyValue);

//@TODO other things with the quantity...

//wait for keypress to continue
Console.ReadKey();
```

Figure 4.31: Validation routine for a defined range in C#

Of course, an input of 0 would indicate that the customer is not trying to purchase any number of the item. However, it is still a logically valid input and would pass validation.

Ranges do not have to be limited to numerical input. For example, the characters 'A' to 'F' could also represent a valid range.

Length check

A length check assesses how many characters have been entered. Some very well-known inputs have limited lengths. For example:

- short message service (SMS) texts are limited to 160 characters
- tweets were originally limited to 140 characters (derived from SMS length minus 20 characters for the user's unique address).

Figure 4.32 shows a C# extract which limits character input to a predetermined maximum length set by a constant.

```
const int MAX = 10;      //set maximum number of characters

string message;          //our message to input and process
int messageLength;       //our message's length in characters

//perform loop while string too long!
do
{
    //input string
    Console.WriteLine("Enter message (max {0} characters)", MAX);
    message = Console.ReadLine();

    messageLength = message.Length;

    //check its length
    if (messageLength > MAX)
    {
        Console.WriteLine("Sorry, your message of {0} characters is too long.",messageLength);
    }

} while (messageLength > MAX);
```

Figure 4.32: Validation routine for a typical length check in C#

Presence check

A presence check assesses whether data is present – that is, whether it exists. For example, a user has to input whether they are 'male', 'female' or 'would prefer not to say' when completing a registration input form. The presence check validates that they have not left the entry blank or unselected.

Type check

A type check assesses whether the data entered is of the correct data type, as in Figure 4.33. For example, if the user has to enter their age, this would require the input to be an integer (a whole number). If input of incorrect data types is not prevented by the programmer, it can cause a fatal run-time error.

```
Age? A           (data type is character, this is INVALID)
Age?  16         (data type is integer, this is VALID)
Age? 01/09/16    (data type is date, this is INVALID)
```

Figure 4.33: Example of a type check.

Format check

A format check assesses whether the data entered is in the correct format, for example, it checks that a string containing a UK postcode follows the format 'PO1 3AX'.

- PO is the area, such as GL (Gloucester) – this has to be capitals, alphabetic, 1 or 2 characters.
- 1 is the district, usually between 1 and 20 per area – this has to be up to 2 numeric digits.
- 3 is the sector, usually covering up to 3000 addresses – this has to be 1 numeric digit.
- AX is the unit, usually covering up to 15 addresses – this has to be capitals, alphabetic, 2 characters.

A format check on a postcode would apply these rules, as shown in Figure 4.34.

```
Postcode? GL2 4TH    (postcode is VALID)
Postcode? W1A 1AA   (postcode is VALID)
Postcode? GL2 24TH   (postcode is INVALID; sector can only be 1 numeric digit)
```

Figure 4.34: A format check on a postcode

Check digit

Typically, a check digit is a single character (usually a numeric digit) derived from an algorithm which is performed on a piece of data. The algorithm is designed to only generate this particular digit if the data (e.g. a string of characters) has exactly those characters and they are arranged in that specific order. Any incorrect character or swapping of character positions generates a different check digit value and fails the test. Check digits are mostly used to detect errors in inputted values such as barcodes, bank account numbers and software registration codes.

An ISBN-10 (10-digit International Standard Book Number) uses a check digit. For example, the previous version of this very book had an ISBN-10 of 1846909287. The last digit (7) is treated as the check digit. You can determine if this code is correct using the follow technique.

Key term

Modulus division – performing a division operation and returning the remainder rather than working out the decimal or fractional answer.

Step by step: Check digit validation

3 Steps

1 Multiply each digit by smaller and smaller positional weights, as shown in Table 4.14.

Table 4.14: Check digit for student book ISBN

Digits in ISBN	1	8	4	6	9	0	9	2	8	7
Factor of multiplication	×10	×9	×8	×7	×6	×5	×4	×3	×2	×1
Subtotals	**10**	**72**	**32**	**42**	**54**	**0**	**36**	**6**	**16**	**7**

2 Add up the subtotals in all columns:

10 + 72 + 32 + 42 + 54 + 0 + 36 + 6 + 16 + 7 = 275

3 Perform **modulus division** by 11:

275 MOD 11 = 25 remainder 0

Is there a remainder of 0? Yes, there is, which means that the ISBN is valid. Any other result means an incorrect ISBN has been entered

This technique identifies incorrect or transposed digits in the code. In the example given in the Step by step, the last digit (7) is considered to be the check digit.

Spelling

A spelling check assesses whether the words being entered can be found within an electronic dictionary – that is, that they are valid words.

Error handling and reporting

Many modern programming languages have syntax features which are designed to handle run-time errors when they occur. If they did not have these features, the application would crash or lock unresponsively.

A common error handling technique is the use of the 'try...throw...catch' expression that can be found in many languages, including C++, Microsoft C#, Oracle Java and PHP. This error handling technique works by trying an operation, catching any possible errors and (optionally) throwing an appropriate exception. This approach prevents the application from failing the operation in an uncontrolled way, as this would cause the application to crash completely.

An example of this error handling technique is shown in Figure 4.35 by illustrating the dangers of dividing by zero. In this example, the division operation is placed inside a try block, just in case the user has entered '0' (zero) as their second number. This has been done because dividing a number by zero can generate a serious error on a computer platform and may result in a run-time crash. By using the 'try...catch' block we are able to avoid this by displaying the exception that has been caught, which, in this case, is a 'DivideByZero' error.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace test
{
    class Program
    {
        static void Main(string[] args)
        {
            float num1;        // first number
            float num2;        // second number
            float quotient;    // result of dividing first number by second
            string strNumber; // string to temporarily store our input

            quotient = 0;

            //get 1st number
            Console.WriteLine("Enter 1st number");
            strNumber = Console.ReadLine();
            num1 = float.Parse(strNumber);

            //get 2nd number
            Console.WriteLine("Enter 2nd number");
            strNumber = Console.ReadLine();
            num2 = float.Parse(strNumber);
```

```
            //try the division
            try
            {
                quotient = num1 / num2;
                Console.WriteLine("{0} / {1} = {2}", num1, num2, quotient);
            }
            catch (DivideByZeroException e)
            {
                //output the handled exception
                Console.WriteLine("Error exception caught was: {0}", e);
            }

            //wait for key press before finishing
            Console.ReadKey();
        }
    }
}
```

Figure 4.35: A Microsoft C# example of error handling using try and catch blocks

Error reporting is an important aspect of software development. It may be directed towards the user, by telling them that they have made a mistake, or towards the developer, so that they can understand where a program has encountered an error and the nature of this error.

Common techniques for error reporting include:

- displaying an on-screen error message and/or error code
- appending the error details to an electronic log file which can be viewed separately
- sending an email to the developer which includes the error details.

Tip

You should always ensure that you do not display debug information on the user's screen. This is because it may accidentally reveal sensitive data or the inner workings of the program. This is particularly important when developing online applications in order to prevent hacking and potential fraud.

Choice of language

There are a number of different factors that influence the choice of programming language for a software development project.

Client preference

Your client may express a preference for a particular programming language.

Research

Investigate why a client may specify a programming language to be used when they are briefing a development team.

Suitability

Depending on the technical nature of the software development task, different programming languages may be more suitable in each case.

- If you were creating an e-Commerce solution, you would typically use PHP (PHP: Hypertext Preprocessor) or Microsoft's ASP .NET technologies. This is because these are server-side scripting languages. They offer the functions, features and libraries that enable the rapid development of this type of solution.
- If you were creating a video game, you might use Microsoft C# ('C sharp'). C# tends to be favoured because of its extensive .NET framework XNA, which enables video game development for Microsoft Windows, Xbox and Windows mobile phones.
- If you were creating program code to interface with electronic hardware, you could use C. These embedded systems make use of C's ability to access devices at a 'bare metal' level, as it can initiate, time and monitor electronic signals with ease in real-time. For this reason, C is a common programming choice for controlling the interfacing of the Raspberry Pi or Arduino with external devices.

Portability

Some programming languages are more portable than others. This means that the program code written can be compiled (translated to machine code or binary) for use on many different CPUs. C is a mature programming language and so countless compilers are available to translate it, including those which cross-compile from one hardware platform to another. For example, a C compiler on a Microsoft Windows platform (X86_64 processor) can create machine code to run on an Android (**ARM CPU**) device such as a mobile phone, tablet etc.

The popularity of Java has reduced some portability concerns. This is because Java code, when compiled into processor '**architecture-neutral**' byte code, can be run in Java Virtual Machines (JVM). These virtual machines exist within a wide range of devices, from mobile phones to set-top TV boxes.

Key terms

ARM CPU – a family of power-efficient CPUs created by ARM, formerly known as Advanced RISC Machines, that are used in a variety of electronic devices such as mobile telephones, tablets and portable game consoles.

Architecture-neutral – code which is not designed to run on a specific family of CPUs but, instead, is run in a virtual machine.

Maintainability

Even though maintenance is one of the later stages of the SDLC, it will influence the choice of programming language. It is unlikely that a programmed solution will never need maintenance. The most common causes of maintenance are bugs and changes to client or user needs. Maintenance can take time, so it is important that the target programming language encourages good developer practices such as readability and extensibility. These factors will result in program code which is easier to maintain, and this is especially useful if the person maintaining the software solution is not the original developer.

Extensibility

Extensibility describes a solution's ability to grow (or scale) as the needs of the client or the users change over time. Some programming approaches, such as object-oriented programming (OOP), are particularly extensible. This is because their class-based nature closely reflects real-world situations, processes and data, ensuring that the code is easy to adapt as needs change. For this reason, OOP languages such as C++ and C# are very popular.

Expertise

The technical ability of the developer is linked to their familiarity with a particular programming language. Unless external 'contract' developers are to be employed short-term for the duration of a project, the choice of language is constrained by available 'in-house' expertise.

Developers are often encouraged to expand their skill set and this means that they may be sent on study programmes by their managers as part of their continuing professional development (CPD). However, companies are often prepared to pay contract developers for expertise in new or in-demand technologies if their in-house developers do not have the required skill set to complete critical projects.

Time

The length of time spent on development is dependent on the complexity of the problem and the choice of programming language and tools used to solve the problem. Most modern development environments are designed to improve the speed at which a programmer can work. This is because improved productivity in the commercial sector is important in reducing costs.

Rapid application development (RAD) is a software development approach that enables programmers to produce code very quickly. This is achieved by using programming languages that allow the programmer to quickly create **prototypes** of the application. These prototypes can then be modified through an iterative process of client testing and review. This can reduce the length of time spent planning the project. RAD also relies on the use of reusable code snippets and drag-and-drop user interface creation. This means that programming languages that support RAD, such as Microsoft's Visual Basic .NET, can have a beneficial impact on development time.

Key term

Prototype – a working model of the desired solution or components of the solution. It may not have full functionality but allows clients and users to test and review the proposed solution. Their feedback then informs the next prototype and the process continues until the final product is ready.

Support

Many programming languages and development tools can be downloaded free from the internet. This allows developers to build commercial solutions with minimal costs. However, there may not be any technical support for these free languages and tools. They also may not be

updated or maintained. For this reason, it is important when choosing a programming language to know that the technology will not be discontinued or become unsupported in the near future. This may mean that a developer may prefer to purchase a development tool that offers full manufacturer support, frequent updates and fixes and a vibrant community of users.

Cost

The development cost of a project can be linked to a number of factors. Many of these factors are directly or indirectly linked to the choice of programming language. Identifiable cost factors include:

- development tools
- programming skills required, either by training existing staff or hiring contract developers with the necessary language expertise
- licences required for languages, tools or distribution of the executable program code
- speed of development (the time taken to develop the software solution in the chosen language)
- ease of maintenance (the likelihood of bugs)
- ease of extensibility.

Calculating the cost of a project is a complicated task, and the choice of programming language plays a key part in cost estimations.

Reflect

Designing any application involves substantial amounts of planning, particularly in terms of investigating and understanding the client's requirements and the complexities of the chosen programming language.

Any proposed solution should not just list the software features you need to consider but should be an exhaustive examination of all the factors which have shaped your solution, including feedback from the client (if you have shared prototype designs) and your peers.

Recording your findings accurately and comprehensively throughout the solutions design (including the preparation of original assets) strengthens your problem solving by providing a solid foundation on which effective development may begin.

Predefined programs and code snippets

Sometimes it is possible to incorporate predefined programs or snippets of existing code within a solution. These must be documented as part of the software solutions design so that future developers know what they are dealing with. Many software development environments offer prewritten snippets of code which can be referenced or simply 'dropped' into a solution while it is being developed. In addition, third-party code may also be downloaded and incorporated with little technical difficulty.

The use of predefined programs and code snippets has its advantages and disadvantages, as shown in Table 4.15.

Developers should always check that their use of a predefined program or code snippet does not breach the terms of use defined by the original developer. For example, a developer may provide their code 'free for use' as long as it is *not* used in a commercially developed solution. If another developer uses this code in a commercial product that they are producing, they would be acting illegally.

Table 4.15: Advantages and disadvantages of using third-party code in a solution

Advantages	Disadvantages
Saves development time.	Code may introduce potential errors if not tested appropriately.
Saves money (if program/code snippets are free and do not require additional licensing).	Code may not be modifiable (in some cases, the original developer may explicitly prevent modification).
Code is likely to be pretested so should be bug-free.	May result in unforeseen compatibility issues.
Code is likely to be written in an efficient and maintainable way.	They may no longer work with the developer's solution if the predefined program/code snippet is updated.
Code may offer additional functionality which may be useful in the future.	Predefined programs may contain malware, especially if downloaded from a disreputable source.
	There may be no support from the original developer.
	They may be discontinued or abandoned.

Ready-made and original assets

Modern programs usually have a media-rich presentation and incorporate a wide range of high-quality assets to enhance their appearance, user interface and functionality. Typical digital assets may include:

- graphics – such as PNG (Portable Network Graphics), BMP (Bitmap) or JPG (Joint Photographic Experts Group)
- animations – such as HTML5 (Hypertext Markup Language 5th major version), Adobe Flash SWF (Small Web Format) and animated GIFs (Graphics Interchange Format)
- audio – such as WAV (wave), MP3 (Moving Picture Experts Group Audio Layer 3)
- video – such as AVI (Audio Video Interleaved), MP4 (Moving Picture Experts Group 4 Part 14)
- **typefaces** and **fonts** – such as Arial, Times New Roman, Verdana.

Key terms

Typeface – the design of an alphabet, i.e. the actual shapes of the letters and symbols. 'Arial' and 'Times New Roman' are examples of typeface.

Font – a word used to describe the digital file that contains the typeface. For example, arial.ttf is a TrueType font file. Many people use the terms 'typeface' and 'font' interchangeably.

Assets used by a program should be included in the software solutions design as a key resource. They are grouped by category and the following details are listed for each asset:

- filenames (and paths)
- file formats
- file sizes
- dimensions/resolution in pixels (for digital graphics)
- duration (for digital audio and video)
- frames (for animation)
- FPS (frames per second – for animation and video)
- notes on their purpose/usage in the program
- any licence information that is required, e.g. original copyright.

Tip

When using any digital asset, you must acknowledge its copyright. Sometimes, you may need to request formal permission to include the asset from the copyright owner. This may involve paying a licensing fee for the copyright owner. In the UK, copyright is defined by the Copyright, Designs and Patents Act 1988.

Feedback from others

A key aspect of the software design process is taking feedback from peers and your client. Designing an application is an iterative process – that is, one that repeats a particular process a number of times until it gets closer to completion. Feedback can be used to refine the next iteration of the process.

Gathering feedback on screen layouts, user interface, navigation, algorithms and so on will help you to refine your ideas. It allows you to identify and remove aspects of the design which do not work well and to identify and keep the ideas that users like. It will also help you to gain confidence in your problem-solving abilities, encourage you to consider alternative design ideas and strengthen your ability to make decisions and justify changes to others.

Link

Feedback is also a key ingredient for the robust testing of mobile apps, as shown in *Unit 7: Mobile App Development*.

Reflect

In the course of developing your software solution, you will have received feedback from your peers, test users and client. The views and opinions of others can be very useful, as they are usually unbiased and can provide a neutral judgement about the design and implementation of your program. This is particularly important when considering weaknesses, as well as strengths.

Learn to take feedback positively. This will ensure that you respond to it in a mature, constructive way and can use it to improve the end product and your problem-solving skills.

Test plan

Testing should form part of the software solutions design. A test plan should detail how you are planning to test your program, step by step, and it will include several use cases. Each use case tells the story of a user's successful or unsuccessful interaction with the program. Use cases include details of:

- the test data entered
- the operations performed
- the order of the operations performed
- the user's response to the program's prompts.

Test data is usually found in one of three possible states:

- typical/normal – data within the acceptable range that the user would usually enter
- extreme – unlikely data at the edges of the acceptable range
- erroneous – data which should not be entered (e.g. text rather than numbers).

Link

For more about test data, see *Unit 5: Data Modelling*.

Test data should be realistically selected, where appropriate. For example, it can be taken from values provided by the client, especially if you are automating a manual process, such as order processing, for your client.

Technical and design constraints

Your problem-solving skills may be limited by a variety of technical and design constraints. These constraints should be reflected in the software solutions design and may include the following.

- Connectivity – which devices or network connections are required for your application? This is particularly important where special hardware is needed (e.g. for input, storage or output) or if the application has to have an active network connection.
- Memory storage – are there RAM requirements that your program's digital footprint must not exceed?
- Programming languages – has the programming language been preselected by the client? This may happen if they have experience with applications written in particular programming languages.

Reflect

Your solutions design showcases your ability to accurately convey your intended thoughts and ideas to others through the written and spoken word. This is true whether you are corresponding with the client via email, writing pseudocode, compiling interview notes or creating flowcharts to represent complex algorithms.

Your use of tone and language must engage productively with the client and the potential users of the finished program.

You may be asked to present your solution to the client. Your presentation will be more successful if you:

- use a positive and engaging tone
- choose an appropriate level of technical language that your intended audience will understand
- avoid using unnecessary jargon.

PAUSE POINT Can you explain what this learning aim was about? What elements did you find easiest?

Hint Close the book and create a simple checklist of the activities and documentation that you need to assemble as part of a software solutions design.

Extend What are the factors that may affect the choice of programming language?

C Develop a software solution to meet client requirements

Once the software solution has been designed, documented, reviewed and agreed, it is time to implement the design to produce a working application. If your design is detailed, you should find it relatively straightforward to convert it to appropriate code in your target programming language.

Software solutions development

Software solutions development can be a time-consuming process. However, modern development tools have removed many difficulties for both novice and experienced programmers. You will have many options available to you when developing your application. Choosing your development environment is your first step.

Development environment

The software development process uses integrated development environments (IDEs). IDEs help to improve productivity by allowing developers to manage projects and edit, compile, debug and execute code, all from within the same software suite.

Link

For more on IDEs, see *Unit 7: Mobile Apps Development*.

Sometimes, the IDE is specific to the chosen programming language. For example, Microsoft Visual Basic .NET uses the commercial Visual Studio or free Visual Studio Express editions, as shown in Figure 4.36.

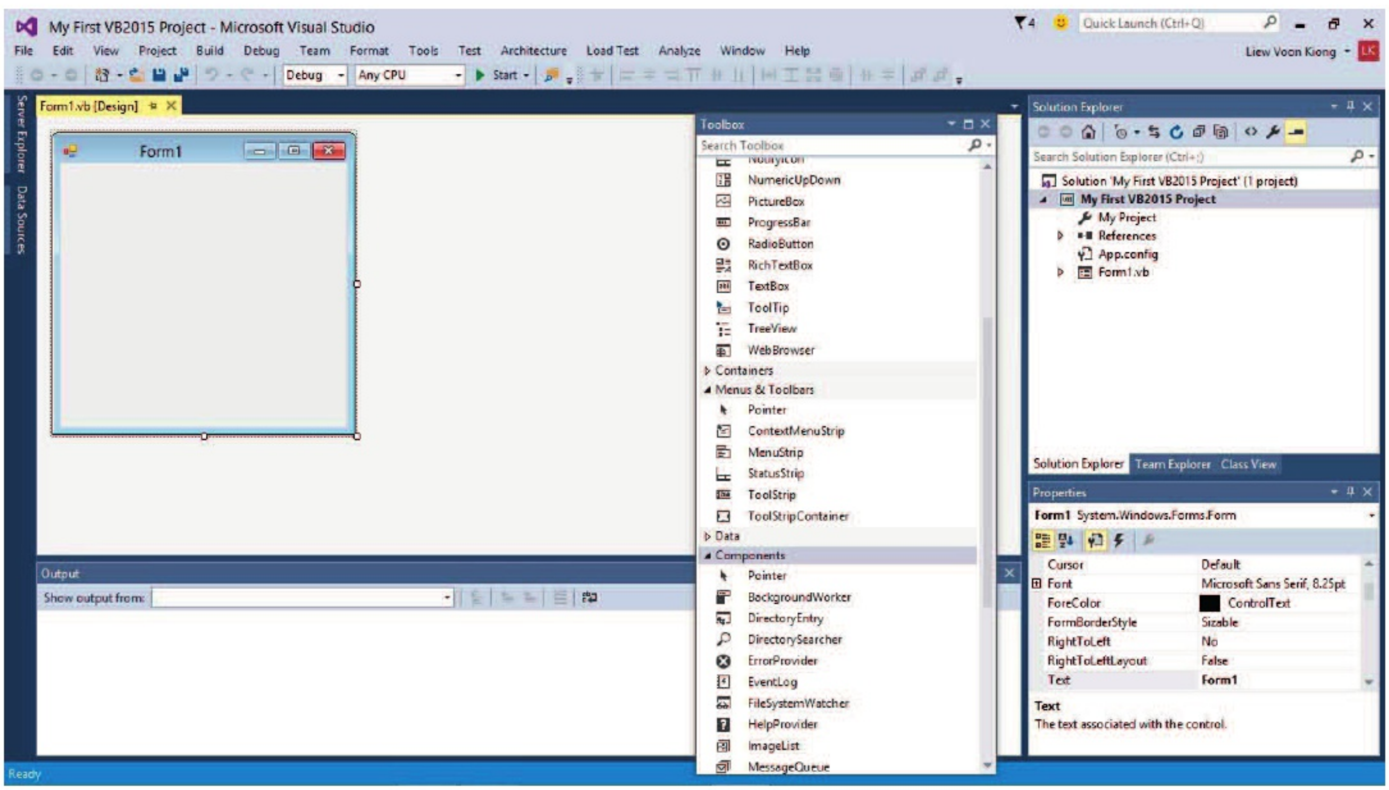

Figure 4.36: Microsoft Visual Basic .NET IDE

Other programming languages do not have a specific IDE, and this may influence the choice of programming language. Popular languages such as C, C++ and PHP can be developed using a variety of development environments. These environments may even be made up of a number of disparate elements. For example, PHP could be developed using either of the following set-ups:

- a Linux operating system, Nano text editor, Apache HTTPD Web Server with PHP5 modules installed
- a Microsoft Windows operating system, Notepad++, Microsoft IIS Web Server with Windows PHP binaries installed.

Either of these solutions would provide a suitable environment for a developer working with PHP.

Link

Common development environments for mobile apps are discussed in *Unit 7: Mobile Apps Development*.

Library routines, standard code and user generated subroutines

The chosen development environment should offer access to library routines, standard code and programmer-defined subroutines. These can be used to make the program more efficient.

Many development environments will contain features to help programmers generate code more quickly. These include the following.

1. Boilerplate code – outline 'skeletons' of standard code which act as a starting point for programmers from which to code their application.
2. Automatic code completion – pop-ups which suggest code that may be used by the programmer to finish their current line of code, often offering a number of different options, complete with definitions and uses.
3. Code snippets – short sections of code which can be dropped into a program from a library of prewritten routines, often including commonly used tasks.

All of these features are designed to save the programmer time and to make the development process more intuitive. In addition, many development environments provide access to online resources containing third-party support. For example, Microsoft's MSDN (Microsoft Developer Network) has an extensive library of resources and active forums.

Link

IDEs are used in all kinds of development. For example, *Unit 7: Mobile App Development* examines the popular IDEs used to create mobile apps for Apple iOS and Android devices and provides a worked example of creating a software application for a specific user need. For more on IDEs in different contexts, see *Unit 6: Website Development* and *Unit 8: Computer Games Development*.

Testing software solutions

As soon as obvious bugs, such as syntax errors, have been successfully removed from a program using the debugging tools available in your programming environment, software solutions can be tested. This involves following a simple step-by-step process, using a test plan and test data.

Worked example: Testing software

Step 1: Ana creates a test plan and selects suitable test data, which includes typical, extreme and erroneous data.

Step 2: Ana feeds her sample test data and user choices into a trace table. She then records the outcomes that she expects to see.

Step 3: Ana feeds her sample test data into the live program and records the actual outcomes.

Step 4: Ana compares the expected and actual outputs to see whether they are different.

Step 5: Ana identifies any discrepancies and revisits the program code to fix the semantic errors that the test has identified.

Step 6: Ana retests the program to see if the changes she has made to the code have fixed the errors. If they have not, she should then repeat the process until the errors are solved.

Other types of testing would include the following.

- Compatibility testing – testing the application within different environments, such as on different operating systems and/or different hardware platforms (desktop PC, Apple Mac, mobile phone, tablet and games console).
- Stability testing – also known as load or endurance testing; this checks that the software solution performs continuously well, time after time, without issues.
- Functional testing – a black box style of testing which ensures that the software solution meets the functional requirements that the client originally presented.

Link

To test software, you need a test plan and test data, which are discussed earlier in this unit. Testing software is also discussed in *Unit 5: Data Modelling* and *Unit 7: Mobile Apps Development*.

Syntax errors are the mistakes in the source code that the programming language's translator cannot pass or resolve, which the translator reports for the developer's attention. These occur when the program code breaks the language's rules or structure, for example when invalid symbols are used or reserved words are incorrectly spelled.

Semantic errors in statements are often more difficult to identify and fix than syntax errors. This is because their syntax is fine, so they are not reported by the translator. However, they are an 'error in logic or meaning', which means that they do not perform the task that you intended.

Tip

Fixing errors in a version of a program (often called a 'build') can unintentionally break aspects of the program that previously worked. Always make sure that you test a program fully every time it is fixed, rather than just testing the aspects that the most recent changes were intended to fix.

Improving, refining and optimising software applications

As a developer, you can use a number of methods to refine and optimise your software solution. These include:

- annotating code, as this will allow developers to repair or debug the program and will improve its maintainability
- program compilation for a designated platform or environment
- reviewing the quality of the program in terms of its reliability, usability, efficiency, maintainability and portability
- undertaking user testing and getting feedback from users on their experience
- making use of the outcomes of testing and feedback in any planned perfective or adaptive maintenance, e.g. using errors found during formal testing to target code which needs fixing or following user feedback to identify aspects which need improvement
- documenting any changes to the design and solution, e.g. changes to program code, required inputs, formatted outputs, data storage etc.

Link

See *Unit 7 Mobile Apps Development*, for a specific example of these methods in developing mobile apps.

Reviewing software solutions

You should always review and evaluate your software solutions once they have been developed. You may find that the following questions are useful when reviewing your solution.

- How suitable is it for the intended audience and purpose?
- Does it meet the client's original needs?
- How user friendly is it?
- Is the software solution of a high quality? Is it:
  - reliable
  - efficient in performance
  - easily maintained
  - portable to other platforms?
- Did the selected programming language contain constraints that compromised the solution?
- Were there any other constraints that affected the solution? These may include:
  - available time for development or testing
  - your personal skills and knowledge
  - issues when using languages on particular platforms.
- What are the strengths and weaknesses of the software solution?
- How could you improve the solution? Think about this in terms of the improvements that you would make in the short term, the medium term and the long term.
- Could the software solution be optimised by:
  - improving robustness, i.e. making the program more resilient against user errors, bad inputs etc. and preventing it from operating erratically or fatally crashing
  - improving efficiency of the code, i.e. making the program execute faster, use fewer resources or simply feel more responsive to the user
  - expanding functionality, i.e. improving the user's experience by incorporating new options, user customisation or adding new tasks for the application to process?

Link

Reviewing is also discussed in more detail in *Unit 7: Mobile Apps Development*.

Reflect

Evaluation is a key skill that you will develop in your work as a programmer. Take what you have learnt from evaluating others' programs and codes and practise evaluating your own. You may find this difficult at first because you will have to look at your own work as though it were someone else's and identify its weaknesses as well as its strengths. However, doing this will improve your programming skills and performance and will develop your ability to make and justify well-informed recommendations. It will also give you more confidence in making decisions as you will be able to analyse your own reasons for making or recommending a particular decision.

Assessment practice 4.2

B.P4 B.P5 B.M2 C.P6 C.P7 C.M3 BC.D2 BC.D3

You are a programmer in a development team, and you have just been introduced to a client who wants your team to develop a bespoke program to meet their company's changing needs.

The client has no fixed ideas about how the problem can be solved or which programming language should be used. However, they have provided a list of the required inputs, a description of the outputs that must be created and the actions that the application must perform.

Investigate this problem and analyse its features. When you think that you understand it to the best of your ability, produce a comprehensive design for a computer program to meet these client requirements.

Ensure that you review the design with others (including the client) and use their feedback to drive improvements to your proposed solution. You should justify all of your design judgements, demonstrating that the solution you have created will be effective and fully meets the client's needs.

Once the design has been signed off by the client, your line manager asks you to implement the design and develop the software solution.

Use a suitable programming language and appropriate features of the chosen development environment to produce the code required. In addition, you need to select library routines and design subroutines to improve the efficiency of your program.

Create and implement a sensible test plan to thoroughly test your program. You must check the code's functionality, stability and compatibility. Once testing is complete, refine your solution by annotating it correctly to enable effective repair and debugging.

Review your code in terms of its reliability, usability, efficiency, maintainability and portability. Use review feedback from users to make changes to the solution and document the rationale behind each improvement.

Finally, you should evaluate the design and the finished optimised program, comparing them with the client requirements to see whether you have fully met their original needs.

You should demonstrate individual responsibility, creativity and effective self-management throughout all the stages of the software development project.

Plan

- What is the task? What am I being asked to do? Do I understand the client's needs?
- What is my starting point for this task? What is the most important thing to know?
- How confident do I feel in my own abilities to complete this task?
- Are there any areas I think I may struggle with, especially technical areas? If so, do I know how to research these?
- Are there any resources or worked examples which would help me to complete this task?
- Before I start to write the code, how well do I think my proposed solution is described in the software design?

Do

- I know what it is I am doing and what I want to achieve.
- I will use this as an opportunity to try new techniques and improve my problem-solving skills.
- I can use feedback and review comments to identify when I have gone wrong and adjust my thinking or approach to get myself back on course.
- I know who to ask for feedback on the ideas I have generated.
- I am prepared to optimise my solution based on identified requirements and user feedback.

Review

- I can explain what the task was and how I approached it, justifying my decisions at all stages.
- I can justify the changes I have made based on feedback I have received.
- I can explain how I would approach the hard elements differently next time (i.e. what I would do differently).

Further reading and resources

Books

Pilone, D. and Miles, R. (2008) *Head First Software Development* (O'Reilly Media) ISBN 9780596527358.

Greene, J. and Stellman, A. (2013) *Head First C#, Third Edition* (O'Reilly Media) ISBN 9781449343507.

Hogan, B.P. (2015) *Exercises for Programmers: 57 Challenges to Develop your Coding Skills* (Pragmatic Bookshelf) ISBN 9781680501223.

Websites

Tutorials point: **www.tutorialspoint.com/index.htm**

Codeacademy: **www.codecademy.com/**

THINK FUTURE

Dan Hardy

Junior Programmer in a software development team

I've worked in the team for just over a year. In that time, I've worked on various programs and I've already learnt so much. Some of the programming tasks we tackle are very complex – I've found that I can't just sit at my keyboard and start writing program code without thinking about the solution properly. When I first started, my manager encouraged me to draw flowcharts to break down difficult business logic into simple steps. I find this approach really works, and I end up returning to my flowcharts a lot while I'm working on the next part of the problem.

When I started, I only knew Microsoft C#. I was worried when I was asked to develop in other languages like Java and PHP. After playing around with these languages, though, I discovered they had similar syntax to C#, and many of the C# concepts I knew were easy to translate. I guess you could say I started to identify the repeating patterns!

When I have to describe to my friends what programming is like, I tell them that it's mainly about problem-solving. I spend a lot of my time refining complex problems into simpler ones and trying to spot any patterns in the problems I deal with.

Focusing your skills

Being adaptable

There are many different programming languages used in computing and the industry is constantly evolving. It is really important that you keep your skills and knowledge as up to date as possible. To get a position as a junior developer in a software development team, you have to be able to adapt yourself and your skills to meet market needs and to keep up with current programming trends.

Here are some questions to ask yourself to help you do this.

- Which programming languages are in demand right now?
- What kinds of skills are sought after? Is there a particular interest in specific operating systems or specific types of applications?
- Are there any other skills related to programming that are in demand, such as knowledge of web technologies or relational databases?
- What experience of and exposure to different programming languages is expected of a junior developer? What skills are mentioned in software development job descriptions and vacancies?
- Can I transfer my programming skills to new languages and different development environments?

Getting Ready for Assessment

Nisar is working towards a BTEC National Extended Certificate in Information Technology. For learning aims B and C, he was given an assignment which asked him to design and develop a software solution to meet client requirements. This is Nisar's experience of developing his software solution for assessment.

How I got started

I started out by revisiting the design for my software solution. I had written it specifically to meet the client requirements that I had been given. I'd already reviewed the design with some of the other students in my class, and this helped me to make important improvements to my proposed solution. Now I just had to implement it.

I realised that the quality of my design would determine how quickly I could get started on building the software solution. Luckily, I'd documented the necessary inputs, outputs and processes really well in the design, so I'd definitely recommend taking the time to do this properly. This helped me to set up the identifiers and data files really easily. The algorithms I'd drawn could be converted to selections and iterations in my target language with very few changes needed, and I could convert the pseudocode I'd written into comments to document my software solution.

I also found that looking back on some of the smaller programs that we had developed in class was helpful. They demonstrated how I should tackle some of the trickier challenges like data file handling. By the end of the assignment, I was really glad that I'd kept my notes!

How I brought it all together

Once I'd debugged and documented the program code, I started to review its functions and features against the original client requirements. I identified areas where I had deviated from the original design, although I found that this was mainly because of technical constraints with the programming language I'd chosen. I thought about my reasons for deviating from the design and then justified them.

I included a section on optimisation, noting down all the improvements that I had made to the code in order to improve the user experience and make it as close as possible to the solution that the client had originally requested.
I also improved my report by evaluating the final design and program code against the client requirements, making balanced judgements about how well the final program had met their expectations.

What I learnt from the experience

I realised that building a detailed design was really helpful when I started to produce the program code. I also learnt that looking back at my design frequently was important. This allowed me to refresh my understanding of the client requirements throughout the development process, and this made it pretty easy to make sure that the program I developed met their requirements and was optimised to meet them in the best way possible.

Think about it

- Do you know the difference between reviewing, justifying and evaluating?
- How will you demonstrate your creativity and individual responsibility when designing, developing and reviewing your computer program?